Imperial College of Science
Department of Computing

Using Learning to Predict and Optimise Power Consumption in Mobile Devices.

By
Edward Kreiman

Dr Emil C. Lupu
June 2010

Abstract

Extending the life of a battery in mobile systems has always presented a challenge, many researchers have come up with efficient power optimisation methodologies. The complexity of modern systems is growing rapidly and such systems cannot be considered as single devices anymore; they must be observed as a system of inter-connected and inter-dependent devices. Applying a single power optimisation technique on a particular sub-system may not be sufficient, it may even have the opposite effect of wasting energy rather than saving energy. The power of the system must be managed entirely and a variety of energy saving techniques must be used to ensure maximum efficiency in the consumption of power at any point of time. Another aspect of modern mobile systems, is the power efficiency of applications running, which lies purely with the developer. Unfortunately development environments and frameworks do not provide a facility to profile an application for debugging power inefficiencies. With this in perspective, the goal of the project is to for automated runtime power management and to provide a framework for power consumption testing. To achieve the goal, we present PowerRunner an automated power management software which predicts and optimises energy consumption on mobile devices, using learning and linux kernels and a low level API for runtime power management. Modern linux kernels set a structure to the power management code to facilitate the suspending and resuming of individual system component at runtime. By carefully monitoring and learning the behaviour of different system components used by various user activities, we analyse and estimate the required resources by an activity. In order to maximise user experience, minimise power requirements, and maximise energy saving we decide and apply combinations of various optimisations at runtime. The project proved to save up to 25% power consistently..

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Emil C. Lupu for all the support, guidance and help he has rendered throughout the entire duration of this project. I am extremely grateful for the huge amount of condense and belief that he placed in me to undertake this project to the best of my ability.
Also, I would like to thank to Professor Susan Eisenbach, for her continuous support and encouragement during the whole duration of my study.

Contents

1 Introduction **6**
- 1.1 Project Goal ... 8
- 1.2 Project Context ... 8
- 1.3 Contributions ... 10
- 1.4 Report Structure ... 11

2 Background Information **12**
- 2.1 Physical Layer ... 12
- 2.2 Power Budget ... 13
- 2.3 Power Manageable Components ... 14
- 2.4 Energy Saving Problem ... 15
- 2.5 Dynamic Power Management ... 17
  - 2.5.1 CPU Dynamic Power Management Algorithms ... 17
  - 2.5.2 I/O Devices Dynamic Power Management Techniques ... 17
- 2.6 Advanced Configuration and Power Interface (ACPI) ... 19
  - 2.6.1 Power States ... 19
- 2.7 Linux Runtime Power Management ... 21
- 2.8 Power Management on Android ... 23
  - 2.8.1 Wake Locks ... 24

3 Power Management Framework **25**
- 3.1 Architecture ... 26
- 3.2 The Observer ... 29
- 3.3 Policy Manager ... 30
  - 3.3.1 Activity ... 30
  - 3.3.2 Policy ... 31
  - 3.3.3 Learning ... 32
- 3.4 Power Management Controller ... 32
- 3.5 Summary ... 33

4 Power Model **34**
4.1 Environment Setup 35
4.1.1 Android Architecture 35
4.1.2 Hardware Architecture 36
4.1.3 Current Measurement Setup 38
4.2 Building Power Model 39
4.2.1 Choosing Variables 39
4.2.2 Current Measurement 41
4.2.3 Logging Session 43
4.2.4 Consolidation and Analysis 44
4.2.5 Linear Regression Model 44
4.3 Results and Validation 46
4.3.1 Summary 47

5 Implementation **48**
5.1 The Observer 48
5.1.1 Hardware Monitoring 49
5.1.2 Monitor probe interface 49
5.1.3 CPU Monitor 50
5.1.4 Screen Monitor 53
5.1.5 Audio Monitor 53
5.1.6 Battery Monitor 54
5.1.7 Bluetooth Monitor 54
5.1.8 GPS Monitor 55
5.1.9 Network Monitor 55
5.1.10 Wifi Monitor 57
5.1.11 Phone Monitor 57
5.1.12 SD Card Monitor 57
5.1.13 Applications Monitor 59
5.2 Power Controller 59
5.2.1 CPU 59
5.2.2 LCD Screen 63
5.2.3 Wireless Network Interface 63
5.2.4 Telephony Interface 64
5.3 Power Policy Manager 65
5.3.1 User Activity and Application 65
5.3.2 Screen Brightness 66
5.3.3 CPU Dynamic Frequency Scaling 67
5.3.4 Network 68
5.3.5 Wireless LAN 68
5.4 Summary 68

6 Evaluation **70**
6.1 Test Setup 71
6.2 Results 72
6.3 PowerRunner as testing tool 73

7 Conclusion **75**
7.1 Review of Goals and Achievements 75
7.2 Limitation 76
7.3 Future Work 76

Bibliography **77**

Chapter 1

Introduction

In the last decade, technology has changed lives dramatically and the most valuable asset now is information. Internet has made it easier for people to communicate, read and publish news, listen to music and watch videos online, shop and trade, publish blogs, monitor health, check tube status and live arrivals/departures, navigation and the list goes on infinitely. Such changes in world perception naturally resulted in a need for information to be accessible everywhere and anytime. Networks became faster and wireless, vendors set up hotspots, cell phones are increasingly replacing landlines, mp3/mp4 devices are replacing music centres and traditional video players. Sales of laptops and net books are growing exponentially every year and are overtaking desktop computers. New devices are introduced quarterly, such as mobile phones, handheld computers, mp3 players, digital cameras, etc. As technology progresses handheld devices become more powerful and converge various functions into a single device. Nowadays platforms such as the IPhone and Android phones can do almost anything from making phone calls, browsing the web, emailing, instant messaging, playing and recording videos to playing augmented reality games. Those platforms have a set of built in sensors, easy to use development frameworks that make it easy for programmers to develop any kind of software. The possibilities are infinite. Unfortunately battery technology is evolving at a much slower pace than the complexity and power requirements of these modern devices. Thus, the need for intelligent power management is a very relevant subject the importance of which is often understated. The main challenge is no longer the complexity of what a device is capable of, rather to keep it functional for a longer time. This project aims to design and implement an ACPI based power management system for a mobile platform. The system that learns the behaviour of the user, analyses power consumption of various activities, predicts the required resources, and applies intelligent power management methodologies to maximise energy saving resulting in a longer battery life.

The field of power management is extensively being researched (Chapter 2), but one of the most important aspects is often ignored and that is user behaviour. It is on a first place personal preferences and wishes defining mobile usage. By observing the behavioural tendencies of users, we learned that users have unique resource requirements for an their individual satisfaction. One prefers playing games or watching videos in full screen brightness mode, while others would be satisfied with 60-70%. However, both sets of users would require a CPU to be in high frequency mode while playing games, and probably would not care or even notice if CPU speed is scaled down while reading an e-book for example or whilst listening to music.

We could exploit the fact that users often do not need all components of the system at once for a particular activity. No particular activity requires all components of a mobile device at any given time Therefore a significant amount of energy can be saved by the use of minimum and necessary resources required by an activity. In order to manage mobile systems resources intelligently and efficiently, we need to recognise activities at runtime, learn usage patterns, and apply relevant power optimisation techniques. Applying optimisation in a multitasking environment such as the Android platform is a challenging task because it is difficult to identify minimal requirements for each application whilst they run in parallel.

In reality, even the most efficient power management, might not protect a mobile system from a high power consuming application from exhausting the battery. It is easy to write an application which uses resources unreasonably, for example extensive usage of a network (sockets are not closed, when no longer needed), or extensive CPU cycles which could be minimised with more efficient algorithms. There is a need to provide tools that can help developers to analyse the power management concept and profile. The Software Development Kits provides profiling tools for analysis of CPU, disk, network and memory usages, but are lacking tools which analyse energy consumption of an application. By making such profiling available for developers, the applications will be more energy efficient and as a result would run longer on exciting hardware. It is essential therefore to have efficient power management of existing applications on mobile devices and new applications being tested for power consumption before implementation.

1.1 Project Goal

In our project, we aim to achieve two major goals.

- The first is to design intelligent modular power management architecture suitable for any mobile platform. It will be able to learn how each particular mobile device is used. Analyse gathered data and define the most efficient power management strategy for that particular mobile device. As a result, a consumer will receive a customised power management solution best suitable for his/her needs.
- The second is to create a generic testing framework, which will enable developers to test their applications for power consumption. The framework will analyse what internal devices are used by the application. Then, by using a linear regression based model for estimation of power consumption, it will transfer information into power units. It will help developers to height leaks and possible savings of power.

The core architecture is based on the Advanced Configuration Power Interface (ACPI) specification (Chapter 2). In this project, we are using Linux kernel implementation as proof of the concept. The framework, however, could be easily replicated on different operating systems that support runtime ACPI. We have chosen the Android OS as a mobile platform for our implementation because it runs Linux and provides access to some of the power management functions through an API.

1.2 Project Context

This project is inspired by the need of an intelligent power management solution for modern mobile platforms. At present, mobile platforms do not provide a power management framework that is focused on user needs or would customise power consumption according to a particular mobile user's profile/consumption. Energy consumption of a mobile device can be adapted to its user. As an example, consider a case where a user does not need constant wi connection, whilst reading his favourite novel, or he doesnt need a 3g connection if his usual emails are short text (2g will be sufficient), or even not constant 2g data connection, if he gets 1-2 emails a day. Obviously, missing or delaying delivery of an email is not an option, therefore by learning the behaviour of a user, or more precisely the applications, the applications either run in the front or as background services we can tune and predict the required hardware to service the users needs and at the same time save energy. The ultimate goal is to manage energy efficiently, operate the mobile device without charging for longer period of time thereby maintaining the experience of the user. In other words our project aims to run as a background process, monitor the system (with minimal overhead), automatically manage the power of subcomponents in the system by applying relevant optimisation techniques (which will be discussed in detail in Section 5.3)

and as a result reduce the overall power consumption of the system.
As mentioned above, the second important area is creating applications with optimum power consumption. It is important to promote and advocate the power management awareness within the emerging community of developers for mobile platforms. This community is growing rapidly as does the number of applications produced. Usually, a developer will write 1-3 applications before he gets enough experience to pay attention to a subject of power management. An application which uses resources avariciously and inefficiently (such as requesting a large file from the internet, when the system is on a 2g connection, or running complex algorithms, or even declaring local variables unnecessarily) could drain the battery very quickly.
In our vision, if developers have tools for analysing power consumption for their applications, as part of the QA process, they would put more effort into writing efficient code thereby creating power efficient applications. An example of such a tool is PowerRunner. It runs in the background, monitors running applications, hardware components, and the resources used by each application. PowerRunner includes an accurate calculation model to estimate power consumption in real time. It is a fundamental part of the testing framework. PowerRunner assists with detailed analysis of the energy use by applications and resources (cpu, network, screen, etc.) and helps to identify potential leaks. The process has an option to run with optimisations turned on. For each known application PowerRunner tunes the system to operate with minimum resources required and disables the unnecessary resources when appropriate (The details of optimisations will be discussed in Chapter 5). As we mentioned earlier PowerRunner is designed on top of a standard ACPI specification. Google's Android platform is running on top of Linux OS and implements a lightweight ACPI driver optimised for embedded systems. Furthermore android exposes some of it through a java API. According to the Linux documentation, most of the Linux power management code is driver specific. There are two models for device power management:

- System sleep - this is a system-wide low power state, akin to 'hibernate' on a desktop.
- Runtime power management - various drivers can enter individual low-power states while everything else is still running.

The actual implementation of these states appears to be system or device specific, so there will be implementations that relate directly to a particular device. This project is implemented on Android Dev 1 device, running Android OS 1.6. However modular design of the architecture allows an easy adaptation to other platforms.

1.3 Contributions

The list below summarises the main contributions of this project:

- Instrumentation framework. This is a modular and extendable mechanism for collecting various types of information from running subsystems. It is used to construct the power model and monitor the consumption of the system in real time. A set of monitors was designed to cover the majority of hardware components on a mobile device. Some monitors use the Android API to retrieve information about components like audio, or GPS, and some collect data directly from the drivers of the components (The monitors are detailed in Section 5.1). The framework allows easily implement and adds new monitors to the instrumentation array.
- An accurate power model. It precisely estimates power consumption of the system as a whole, which is broken down into individual components of the system. To build the model we ran a series of tests on the device, using different combinations of subcomponents, in conjunction with non-intrusive current measurements. We took these measurements with the highly sensitive multimeter FLUKE 289. Once we collected around 10,000 samples, we applied the data mining technique to extract the power coefficients of each component of the device (Chapter 4). This power model is used by testing the framework to identify power leakage and real-time power usage estimation of applications.
- Power estimation engine is for analysing application/activity power requirements. An observer module, whose function is to manage the monitors, aggregate information and pass it to the Policy-Manager. This has an estimation engine which calculates the consumption for an individual probe using the model.
- Intelligent Policy Manager. It learns the behaviour of a user and provides heuristic effective power management. Every running application needs to be classified as a type of activity. By identifying patterns in the behaviour of the application we automatically classify it in runtime and apply relevant optimisations for an activity. Users' adjustments are recorded into the knowledge base so the next time the same application is run we apply a custom power policy. This module constantly analyses the state of the system, collected by the observer, and identifies activities (one or more), then executes the minimal power requirements policy through the power controller.
- Power controller. It manages the power state of the device by controlling states of hardware components. Power controller acts as an execution arm for policy manager. It has modular architecture, which provides an infrastructure for custom control modules for hardware components.

1.4 Report Structure

This report is structured as follows:

- Chapter 2 presents the power minimisation problem by showing how a hardware component can be made to use less power followed by an explanation of different strategies for effective power management. Following this, an overview of past research in power management is presented touching on all the different elements affected by power management: the hardware, the application and the operating system.
- Chapter 3 explains in detail the design of our power management framework by presenting the mechanisms used to instrument the hardware components. This is followed by explaining the of learning mechanism in the framework and showing how it can be used to implement power management techniques. Finally, the policy management and hardware controls processes required to manage the system's power consumption are presented.
- Chapter 4 presents the linear regression based model for estimation of power consumption in run-time. Details about the implementation of the model are presented at the beginning of the chapter followed by the derived linear parameter coefficients for hardware components. Finally, the validation of the model is presented.
- Chapter 5 describes how our architectural design has been implemented on an Android platform. Details about the implementation platform are presented at the beginning of the chapter followed by a description of the implementation details of each module in our framework and their interaction with each other.
- Chapter 6 presents some tests done to evaluate the correctness of our implementation and the validity of our design. Tests performed to observe the power consumption of the system with PowerRunner are first presented followed by the presentation of power-testing framework.
- Chapter 7 summarises the achievements of this project and reviews our initial goals. A discussion on particular problems encountered is then presented followed by suggestions on how this project can be extended through future work.

Chapter 2

Background Information

In this chapter we present an overview of the state of the art eld of power management and the technologies behind PowerRunner. We first discuss the theoretical basics and the rationale of power reduction based on dynamic power management. Then we will explore in great detail some of the techniques and implementations of dynamic power management. In modern mobile devices power management is taken very seriously by the hardware vendors as well as Operating System vendors, as affects core parts of the systems. We will discuss the general concepts of power management, followed by a discussion of the modern techniques to reduce energy consumption and then apply these to the world of mobile devices. We will continue with a discussion of how some of the techniques t into the PowerRunner design, and how it will impact on energy saving In mobile devices.

2.1 Physical Layer

On conventional mobile systems and systems such as distributed sensor networks the main source of energy is of course the battery. The lifetime of a battery is an important variable in system performance. Users and system designers effectively face a trade off between system performance and lifetime of the system.

Power is dened as the amount of work done in a given period of time and is measured in Watts (1 Watt = 1 Joule per Second). An Electromagnetism Watt is defined as the rate at which work is done, when 1 Ampere of current flows through an electrical potential

difference of 1 volt (V).

$$W = VA \tag{2.1}$$

Considering a battery has limited capacity and the amount of energy is also limited, only the time this energy can be used is variable. Therefore to maximise the lifetime of the battery, the power used by the system needs to be minimised.

The basic capacity metric was defined by Peukert‘s law in the 19th century:

$$t = \frac{Q_p}{I^k} \tag{2.2}$$

where:

- Q_p is the capacity when discharged at a rate of 1 amp.
- I is the current drawn from battery (A).
- t is the amount of time (in hours) that a battery can sustain.
- k is a constant around 1.3 - defined by the chemical family and battery design

Using this basic imperial we could relate battery lifetime to other factors in the system.

2.2 Power Budget

Battery capacity is measured in mAh. Typical mobile devices have batteries with capacity of 1150-1500 mAh. Once we know the systems power consumption, we can easily calculate a discharge rate, and the limitation of the current battery lifetime.

Mobile devices can be viewed as a collection of heterogeneous components typically composed from a digital large scale integration board (VLSI), radio-frequency board, memory card, screen, GPS receiver and so on. Those components can be active at different moments of time, and proportionally to the workload will consume different fractions of the power budget [[1]]

In Figure 2.1 , we can see how energy is being spent in a mobile device

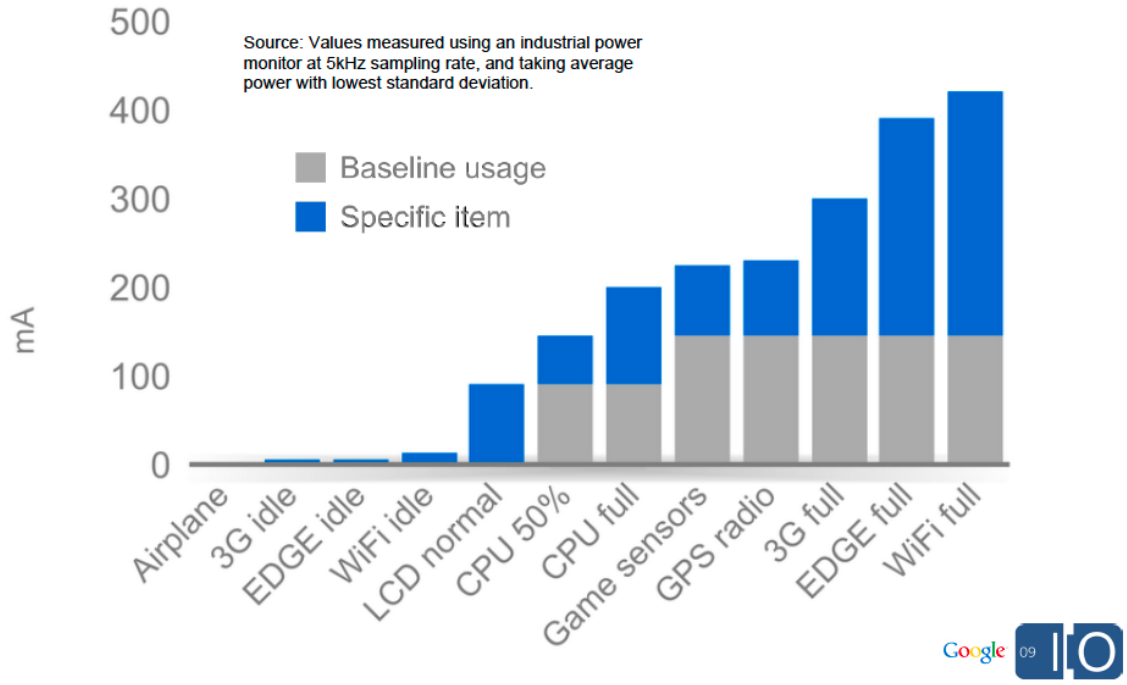


Figure 2.1: Where does it all go ? [2]

In real usage those figures translates directly into the time the device is on. Here are some examples [2]:

- Watching YouTube: 340mA = 3.4 hours
- Browsing 3G web: 225mA = 5 hours
- Typical usage: 42mA average = 32 hours
- EDGE completely idle: 5mA = 9.5 days
- Airplane mode idle: 2mA = 24 days

2.3 Power Manageable Components

A Mobile device can be viewed as a collection of heterogeneous components. Benini et al. in [3] proposed a model for power managed system as a set of power manageable components

(PMC's) controlled by a power manager (PM). The model defines a component as an atomic block in a complete system. At the system level, component viewed as a functional unit. The fundamental property of a PMC is the ability to operate in different modes, which span the power-performance trade-off. Components which are not managed, and don't operate in multiple performance modes are usually explicitly designed for a targeted performance, and power cost.
Typically PMC's are designed with only few operation modes to reduces the complexity and the overhead of the power management. However the flexibility PMC provide, have a non negligible cost which must be taken into account. In particular the cost of the transitions between the modes, in terms of performance loss, delay, or even additional power (when components need to be initialised, and stabilised).
The PMC can be modelled by a finite state machine (PSM). Each state represent an operation mode, and has attributes of performance and power cost. State transitions has a power and delay cost. Intuitively low-power states, have lower performance and higher transition delay the high-power states. (e.g an idle disk need to spin up to 7200 rpm, before it can operate, but spin down and become idle almost instantaneously). Many single chip devices like CPU and memory as well as more complex devices such as disk drive, wireless network cards, LCD screen and more, can be presented in this simple and abstract model [4, 5, 6].

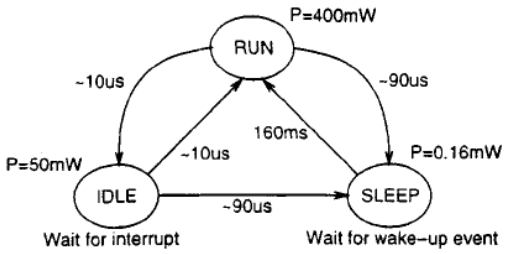


Figure 2.2: Power state machine for the StrongARM SA-1100 processor. [3]

2.4 Energy Saving Problem

The basic idea behind power management is *idleness exploitation* [7]. Whenever device (or component of device) is idle, its energy consumption should be reduced to minimum. To simplify the problem consider a trivial case where PMC have two power states (ON/OFF) and transitions between the two instantaneous; the power and performance cost to perform a transition is negligible. In such case the task to save energy is trivial: as soon as the PMC is idle, it can be transitioned to the lowest power state available. On the arrival

of a request, the PMC is instantaneously transitioned to active state. Unfortunately, in a real system, naively switching off idle devices is not efficient or could even be counter productive in terms of power cost. In some cases returning from low-power state requires a time for:

1. tuning and stabilising the power supply
2. reinitialising (e.g. wifi card will be required to reconnect to the network, hard disk to spin up, etc.)
3. restoring the context

Since typically transitions between power state do have a cost, the optimisation is a difficult problem. In other words, the challenge is to decide when (if at all) its efficient to transition to a low-power state. Benini et al. in [3] suggests that PMC can be transitioned to low-power state S when the time of inactivity T_n is greater then the minimum inactivity time required T_{BE} (*break-even time*) to compensate the cost of entering the state. Where inactivity time T_n is the time PMC spends in low-power state and the time take to enter and exit it. Thus, $T_n > T_{BE}$ where T_{BE} is the sum of the total transition time and the minimum time that has to be spent in inactive state to compensate the additional transition power P_{TR} :

$$T_{BE} = \begin{cases} T_{TR} + T_{TR}\frac{P_{TR}-P_{On}}{P_{On}-P_{Off}} & P_{TR} > P_{On} \\ T_{TR} & P_{TR} \leq P_{On} \end{cases} \tag{2.3}$$

Where,

- T_{TR} - Transition time
- P_{TR} - Transition power
- P_{On} - Power at the ON state
- P_{Off} - Power at the OFF state

Therefore to minimise the power-performance trade-off and save energy, all is left is to know whether T_n will be greater than T_{BE} before we initiate a transition to low-power state. Many authors proposed schemes for dynamic power management (Section 2.5), which falls into two groups of strategies for this problem. The first approach is to wait a *timeout* threshold, and then switch OFF the device. The second approach is to predict the value of T_n and then make a decision based on the prediction.

In our project we addressed the problem by learning the behaviour of user, and PMC's behaviour under workload of user's applications, then in turn applying a suitable power

policy for each of the PMC's.
For example, we observe a user running a mail client which periodically in background pulls mail messages, and typical network activity is low (due to length and number of the messages). Our policy manager algorithm will switch off the network cards (Telephony, and WIFI), periodically will check for known wifi hotspot (location based), if no known networks available it will switch on 2G connection (GPS/EDGE), and wait until it will finish the network activity. However, perhaps over time the network activity will become higher (due to increased number of messages, or their lengths), then the policy manager will adjust the "down" period and will use 3G connection to reduce network activity time. A totally different policy is a applied to network cards, under activity of web browsing, by learning users behaviour we have an ability to decide more precisely if, when and for how long to switch the network interfaces off. We cover the implementation of policy management topic in detail in Chapter 5.

2.5 Dynamic Power Management

In the recent years a great amount of work was put into research on low-power design techniques. Those researches mainly concentrated on reduction the power consumption of the CPU and the I/O devices.

2.5.1 CPU Dynamic Power Management Algorithms

In Yao et al.,[8] the authors present an offline preemptive task scheduling algorithms to minimise energy consumption. This technique search for critical intervals (intervals in which a constant maximum voltage is required to run a group of tasks in optimal schedule), which are scheduled using the EDF scheduling policy. An alternative approach proposed in Ishihara and Yasuura,[9] the method is to use integer linear programming formulation and statically assign voltages to tasks. Authors showed that energy is minimised only if task finishes on scheduled deadline and at most 2 voltages are required to emulate an ideal voltage level. Another online DVS method based on the rate-monotonic algorithm(RMA) is presented by Shin and Choi,[10]. The method identify time frames when processor speed can be scaled down without missing any task deadline. Majority of those techniques are based around the idea of identifying online or offline (applying scheduling algorithms) time instances, where CPU can be scaled down.

2.5.2 I/O Devices Dynamic Power Management Techniques

DPM for I/O devices fall into three categories: timeout, predictive and stochastic.

Timeout Based Techniques

The most basic and widely used techniques are timeout based. The idea is very simple and efficient. The System shut down an I/O device if its being idle for more than specific period of time [3]. The next task requesting the devices wakes it up, and the device process the request. Although the technique is simple and relatively efficient, it has some downsides: Usually the request arrive while the device is off, suffer from a significant delay (a noticeable transaction time from off to on state) While the system countdown the threshold time for a specific device, a great amount of energy is wasted.

Predictive Techniques

Predictive techniques are more readily adaptive to changing workloads than timeout based. They aim to reduce wasted waiting time and eliminate the off-to-on state transaction delay time. Predictive methods by observing the past requests, predicts the length of the next idle period, so the device can be put in low-power state as soon as the period starts. [11, 12]

Stochastic Techniques

Stochastic methods aiming to model the devices requests through different probabilistic distributions and solving stochastic models to figure out device's switching times.[13, 14]

Machine Learning Techniques

Another approach is to use machine learning technique proposed by Chung et al.. Chung et al. in [15], used a simple decision trees to classify past idle periods, and predict future ones.

Summary

All the DPM techniques mentioned above, suit very well a non real-time systems which are tolerant to small delays in computation as a price for prolonging the battery life. However due to their highly probabilistic nature are not usable in real-time systems, as shutting down a device at the wrong time, can potentially result in task missing its deadline. In real time systems, meeting deadlines is critically important, therefore for such systems more deterministic methods are required to guarantee real-time behaviour.

2.6 Advanced Configuration and Power Interface (ACPI)

In December 1996, Intel Motorola and Toshiba release their first specification (ACPI) of open standard for unified operating system-centric device configuration and power management (OSPM)[16]. Main purpose of this specification is to move away from existing standards such as Advanced Power Management (APM), MultiProcessor Specification (MPS) and the Plug and Play (PnP) BIOS specification, towards centralised power management by the Operating System as opposed to the previous BIOS central system, which relied on platform-specific firmware to determine power management and configuration policy. ACPI is activated by OSPM compatible operating system, and takes full control of power management and device configuration. The OSMP-aware Operating system must expose an ACPI-compatible environment to device drivers , which exposes certain system, device and CPU states.

2.6.1 Power States

Global States

The ACPI specification defines the following seven states (so-called global states) for an ACPI-compliant computer-system [17]:

- **G0** (S0) System is on. The CPU is fully up and running; power conservation operates on a per-device basis.
- **G1** Sleeping (subdivides into the four states S1 through S4)
  - **S1**: All processor caches are flushed, and the CPU(s) stop executing instructions. Power to the CPU(s) and RAM is maintained; devices that do not indicate they must remain on may be powered down.
  - **S2**: CPU is off, RAM is refreshed; the system uses a lower power mode than S1.
  - **S3**: Commonly referred to as Standby, Sleep, or Suspend to RAM. RAM remains powered
  - **S4**: Hibernation or Suspend to Disk. All content of main memory is saved to non-volatile memory such as a hard drive, and is powered down.
- **G2** (S5) Soft Off. G2, S5, and Soft Off are synonyms. G2 is almost the same as G3 Mechanical Off, but some components remain powered so the computer can "wake" from input from the keyboard, clock, modem, LAN, or USB device.

- **G3** Mechanical Off: The computer's power consumption approaches close to zero, to the point that the power cord can be removed and the system is safe for disassembly.

Furthermore, the specification defines a Legacy state: the state when an operating system runs which does not support ACPI. In this state, the hardware and power are not managed via ACPI, effectively disabling ACPI.

Device states

The device states D0-D3 are device-dependent: D0 Fully-On is the operating state. D1 and D2 are intermediate power-states whose definition varies by device. D3 Off has the device powered off and unresponsive to its bus.

Processor states

The CPU power states C0-C3 are defined as follows: C0 is the operating state. C1 (often known as Halt) is a state where the processor is not executing instructions, but can return to an executing state essentially instantaneously. All ACPI conferment processors must support this power state. Some processors, such as the Pentium 4, also support an Enhanced C1 state (C1E or Enhanced Halt State) for lower power consumption,[7]. C2 (often known as Stop-Clock) is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional. C3 (often known as Sleep) is a state where the processor does not need to keep its cache coherent, but maintains other state. Some processors have variations on the C3 state (Deep Sleep, Deeper Sleep, etc.) that differ in how long it takes to wake the processor. This processor state is optional.

Performance states

While a device or processor operates (D0 and C0, respectively), it can be in one of several power-performance states. These states are implementation-dependent, but P0 is always the highest-performance state, with P1 to Pn being successively lower-performance states, up to an implementation-specific limit of n no greater than 16. P-states have become known as SpeedStep in Intel processors, as PowerNow! or Cool'n'Quiet in AMD processors, and as PowerSaver in VIA processors. P0 max power and frequency P1 less than P0, voltage/frequency scaled Pn less than P(n-1), voltage/frequency scaled

2.7 Linux Runtime Power Management

A substantial amount of work has been done in the recent years on power management in Linux systems, much of that work concentrated on suspend and hibernation capabilities. However the true value is in being able to reduce energy consumption of a running system. That is true as for large enterprise servers as it is for mobile devices. Some recent developments introduced an API in the kernel, based on ACPI spec. This API sets a structure to the power management code to facilitate the suspending and resuming of individual system components at runtime. Operating system controls the power states of the devices, through special ACPI drivers. An ACPI driver of a particular device implements three methods: runtime_suspend(), runtime_resume() and runtime_idle() which put the device into corresponding state.

The dev_pm_ops structure is augmented with three new functions [18]:

int (*runtime_suspend)(struct device *dev);
int (*runtime_resume)(struct device *dev);
int (*runtime_idle)(struct device *dev);

runtime_suspend()

These functions are supposed to be implemented by the driver of each device for each bus type; they also can act as bus-specific driver callbacks. The kernel will call runtime_suspend() to prepare a specific device for a lower-power state. However, the call doesn't obey the device itself to be suspended but the device does need to prepare for a condition where it is no longer able to communicate with the CPU or memory. In other words, even if the device does not suspend, hardware between that device and the rest of the system might be suspended. A return value of -EBUSY or -EAGAIN will abort the suspend operation.

runtime_resume()

When a method runtime_resume() is being called, the driver should prepare the device to operate again with the rest of the system. the driver should power up the device if needed, restore registers, and everything else required to get the device fully functional.

runtime_idle()

Kernel is calling runtime_idle() on a devices driver, when it thinks the device is a idle and might be a good candidate for suspending. The callback should decide whether the device can really be suspended

Part of the Linux kernel's power management API also a set a functions responsible for suspend and resume activities, deal with mid-course cancelations, external code making

modification in the power management and so on.

Another aspect of this API is the ability to invoke suspend and resume callback asynchronously, which allow them to run in parallel. As long as there are no dependencies between a pair of devices, suspending or resuming them in parallel makes full-system transition faster. However, a problem rises when there are dependencies, running a set of power management operations in parallel might mess up the order. To overcome this issue a completion object is added to each device. In the next section well discuss how Android platform leverage this infrastructure, and how this can be exploited for optimisation purposes.

2.8 Power Management on Android

In this project we used Android as proof of concept. Android support its own Power Management implemented on top of standard Linux RunTime ACPI infrastructure. The main design concept is that CPU shouldn't consume power unless applications services require power.

The Android framework requires the application and services to request a CPU resource with a wake locks, through the application framework and native libraries. If the OS doesn't find any active wake lock, it shut down the CPU. In figure 2.3 shown an infrastructure of Android power management.

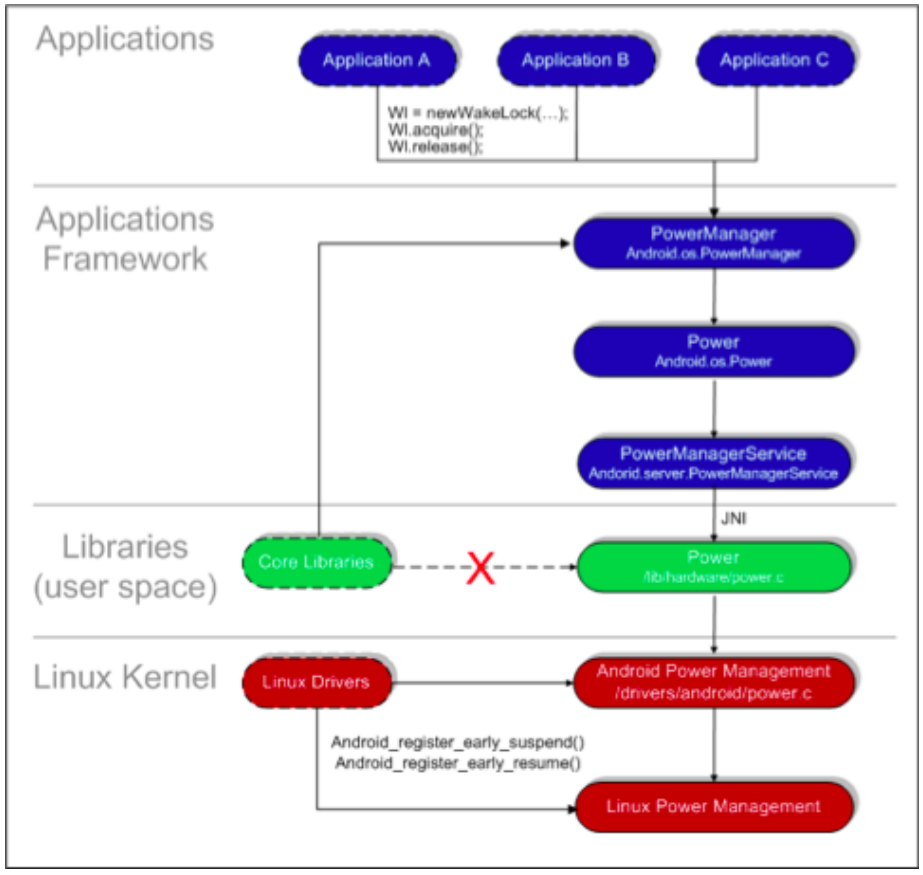


Figure 2.3: Android Power Management Architecture

Android framework exposes seven different types of wake locks for applications and services. For example : 'keep the cpu running, while the screen is off'. Developers choose the most appropriate lock for their applications. Incorrect usage of wake locks, might result in great

power waste.

2.8.1 Wake Locks

- ACQUIRE_CAUSES_WAKEUP

  Normally wake locks don't actually wake the device, they just cause it to remain on once it's already on. Think of the video player app as the normal behaviour. Notifications that pop up and want the device to be on are the exception; use this flag to be like them.

- FULL_WAKE_LOCK

  Wake lock that ensures that the screen and keyboard are on at full brightness.

- ON_AFTER_RELEASE

  When this wake lock is released, poke the user activity timer so the screen stays on for a little longer.

- PARTIAL_WAKE_LOCK

  Wake lock that ensures that the CPU is running. The screen might not be on.

- SCREEN_BRIGHT_WAKE_LOCK

  Wake lock that ensures that the screen is on at full brightness; the keyboard backlight will be allowed to go off.

- SCREEN_DIM_WAKE_LOCK

  Wake lock that ensures that the screen is on, but the keyboard backlight will be allowed to go off, and the screen backlight will be allowed to go dim.[19]

Chapter 3

Power Management Framework

In this chapter we introduce the high level architecture of the power management framework in PowerRunner.

The basic idea behind power management is *idleness exploitation* [7]. Whenever hardware device is idle, its energy consumption should be reduced to minimum. In this project we analyse and learn user's activity on the mobile device. Each type of activity has unique requirements in terms of hardware resources. A typical application require only a subset of hardware components to be active. The power consumption of idle components (which are not used by that application) can be minimised. The knowledge of user activities allows to decide when to put devices into energy saving mode and to predict for how long. At the high level framework can be summarised as follows. A mobile system contains number of heterogeneous components. The framework monitors active applications and the states of hardware components. Framework learns application's usage of hardware resources. The state information is then analysed. The power management polices are applied in runtime to support current and future resource requirements. The framework then changes to the states of individual components which results in saved energy.

3.1 Architecture

In [20], Benini et al. proposed a basic structure for power management framework. As illustrated in figure 3.1, the structure contains three fundamental parts:

- An Observer - module responsible for monitoring the system's hardware.
- A Policy manager - module responsible for managing the power states of the system's components (eg. CPU speed=low, network state = off, screen brightness=135).
- A Controller - module responsible for interaction with the hardware and for execution of state transition commands.

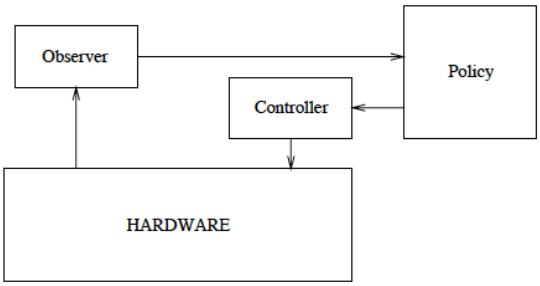


Figure 3.1: Structure of the power manager [20]

We adopted this simple design in our architecture, and extended the Observer and the Policy Manager modules. In figure 3.2 a system power managed by PowerRunner is shown.

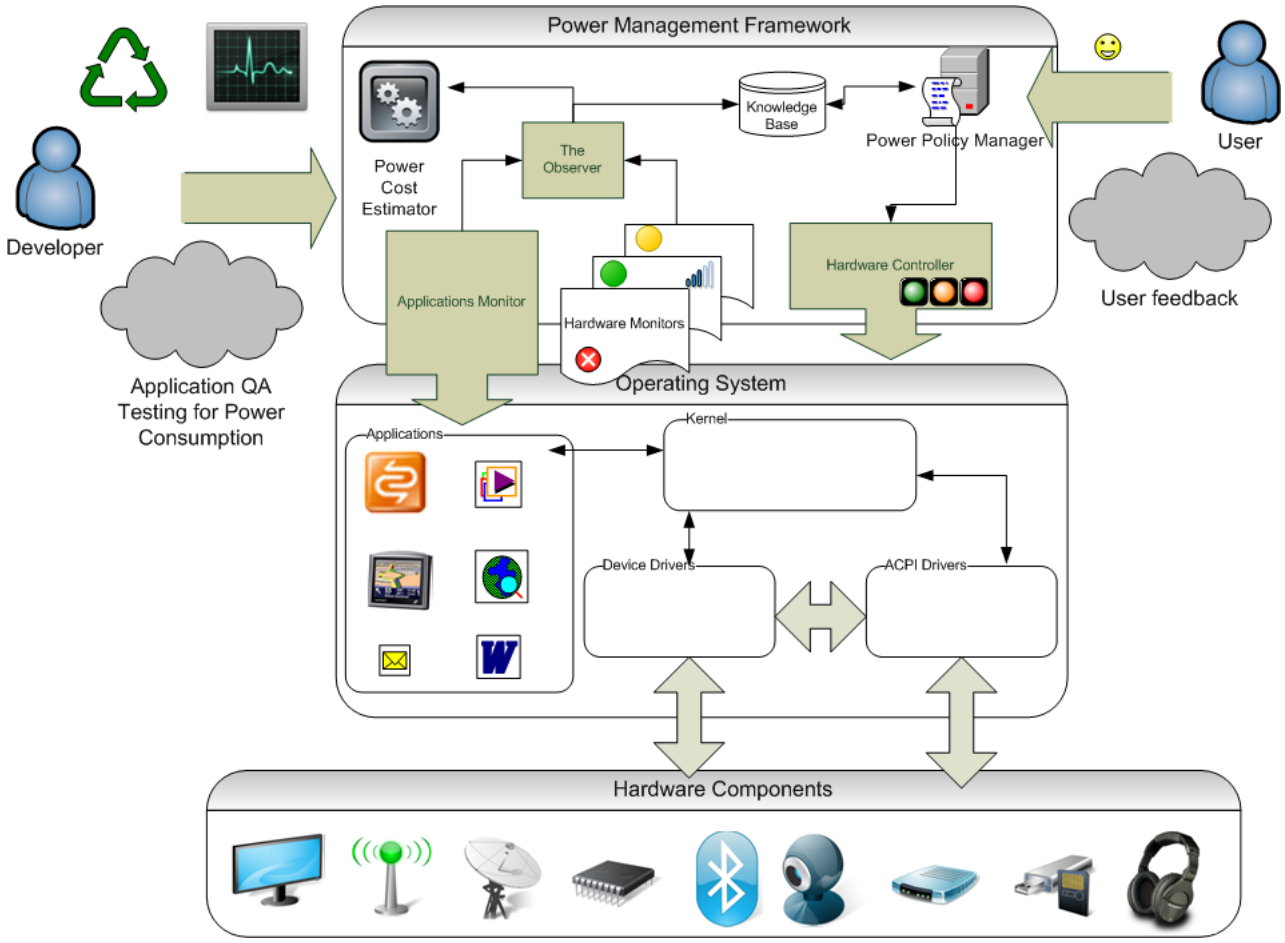


Figure 3.2: Architecture of the PowerRunner

At the lowest level mobile platform have of a collection of hardware components. Those components are controlled by the Operating System's Kernel. The Kernel control the devices by means of drivers[1]. An operating system also provides an ACPI abstraction for power manageable components. Operation states of hardware components are controlled through special ACPI drivers. When an application needs to interact with a hardware device, the kernel issues commands to the corresponding drivers.

The top part of the diagram 3.2 is the architecture of power management framework in PowerRunner. The framework consists of the following subsystems:

1. **The Observer (OB)** - module responsible for coordinating the work of monitors: AM and HM.
2. **Application monitor (AM)** - a monitor responsible for collecting information from the operating system about the running processes, and their utilisation statistics.
3. **Hardware Monitors (HM)** - a collection of monitors. Each monitor is responsible for probing a particular hardware device. HM's interact with hardware abstraction layer of the OS. They collect operation states and utilisation statistics of the hardware components in the system.
4. **Power Cost Estimator (PCE)** - function of this unit is to estimate power consumption of individual hardware components in the system.
5. **Knowledge base (KB)** - in this unit the framework stores the information about users's applications. Information includes the following: classification by activity, hardware requirements, average length of operation and estimated power cost. The knowledge base also holds the historic data of system operation and power consumption.
6. **The Policy Manager (PM)** - function of this module is to apply power management techniques to save energy. It applies power management policy on the system, which enables the resources required by user's activity and minimises the consumption of hardware devices that are not required. The PM also learns new policies for specific activities from the user's feedback.
7. **The Power Management Controller (PMC)** - This module executes the power polices by changing the operation modes of hardware components. It interacts with hardware abstraction layer of the operating system, and issues commands to the

[1] *A device driver is a software allowing higher-level computer programs to interact with a hardware device. A driver typically communicates with the device through the system's bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Drivers are hardware-dependent and operating-system-specific.[21]*

ACPI drivers of the corresponding devices. The controller also maintains a scheduler for time based policies.

8. **The User** - The framework provide a facility for user's feedback and control. The user can adjust the power management policy in runtime, by manually changing the operation modes of hardware devices. After that the framework then captures changes, and learns new policies for a specific application.

The operation of the framework can be summarised as follows. At runtime hardware devices and running application are monitored by the observer module OB. Periodically OB takes a snapshot of the system, which contains the probes of hardware devices and a list of active applications. For every hardware probe OB uses PCE to estimate its power consumption. Then OB stores the snapshot in the history KB and notifies the PM. PM analyses the new snapshot of system and decide whether the current power management policy needs to be changed. If PM decides to change the policy, it stores the new policy in the KB and calls the PMC to execute the required changes on hardware devices. Once PMC retrieves a change request for a list of devices, it issues commands to the corresponding ACPI drivers. At any moment of operation, a user can interfere and make changes to the state of the hardware components (e.g increase the brightness, or enable a particular device). PM notices those changes (through the observation mechanism), correlates them with an active application and learns a new policy. Next time the user will run this specific application, PM will apply the newly learned policy.

We now discuss the details and the challenges of each of the modules mentioned above.

3.2 The Observer

An efficient implementation of dynamic power management governance, requires a mechanism for capturing and analysis of both systems workload/performance ratio of hardware components and estimated power cost which will be covered in Chapter 4. The former may vary in complexity and range of the policy regimes from simple timeout schemes to complex statistical models. As we based our power management strategy on user's context, we introduced two types of monitors. One type is responsible probing the hardware devices in the system, while the other for capturing the state of applications running on the OS. Because hardware components are unique in their nature, the observer module has several dedicated hardware monitors (one per each device) and a single monitor for capturing the state of applications. The operation of the monitors is coordinated by a controller, which starts, stops and gathers probe information from the monitors. Periodically the controller collects probes from the monitors, and constructs a 'snapshot' of the system for an interval of time (since the last snapshot). It then estimates the power consumption for the interval of time (broken down by components). And persist the snapshot into the knowledge base.

The communication between the observer and the policy manager is asynchronous. The observer notifies the PM when new snapshot is available in the knowledge base.

3.3 Policy Manager

3.3.1 Activity

We now introduce a concept of *Activity*, we are using in our framework. Modern mobile devices contains large set of components, which provides a platform for a growing range of applications. Applications can be classified by a finite set of activities:

- Books
- Business
- Education
- Entertainment
- Finance
- Games
- Health and Fitness
- LifeStyle
- Medical
- Music
- Navigation
- News
- Photography
- Productivity
- Reference
- Social Networking
- Sports
- Travel
- Utilities

- Weather

Each type of activity has an attributes of system requirements, pattern of operation (Front/Background), and Time of operation. For example music activity would last longer with constant workload in contrast to a Utility activity which has a short operation time (e.g. calculate mortgage). We can use this knowledge for our benefit to help choosing the more suitable power policy. As an example consider a user reading an ebook, while he is concentrated on reading he most likely not using a network, but probably would prefer a high brightness mode of the screen. Applications are classified as type of activities above, and initially inherit the properties. With time, while the applications are running, user can adjust the system components, like screen brightness, CPU speed, GPS, etc. We observe those adjustments and record them in application knowledge base. Of cause in multithreaded operating system such as android, user can run several applications simultaneously, in which case our software will apply the policy to satisfy the application requires greater resources.

3.3.2 Policy

The power policy is essentially a vector of configurations for hardware devices. Each configuration contains three fields:

- **ON/OFF state** - (0,1)
- **A scheduling definition** ([*,*,*,*])
- **Operation parameter** - an extra options field specific per device (String)

The scheduling defining takes a form of CRON expression [21], and specify the operation times of the device:

```
.---------------- milisecond (0 - 999)
|   .------------- second (0 - 59)
|   |   .---------- minute (0 - 59)
|   |   |   .------- hour (0 - 23)
|   |   |   |
|   |   |   |
*   *   *   *
```

For example configuration for Network interface in a form of: {1,[*,*,*/5,*],"300000"}, means switch the network ON every 5 minutes and switch it OFF 5 minutes later. A form {0,[*,*,1,*,*],"idle"} means switch network off after 1 minute of idleness.

The hardware controller takes care of scheduling mechanism and execution of power management policies.

3.3.3 Learning

The policy manager learns new policies for specific applications from user's feedback and application's behaviour. The former type of learning is based on user's satisfaction of performance. For example a policy for a particular application defines the screen brightness to be 155/255 units (60% of maximum). A user can adjust the screen manually, while in background the policy manager will learn this. Next time the user runs the same application, policy manager will apply the preferred screen brightness. The other type of learning is by analysis of historic activity of a component. Consider a case when a news agent runs in background and periodically pulls news updates from the Internet. The application is configured to check for updates every 5 minutes. However updates usually published every 10-15 minutes (on this news server). By analysing the history of network card activity for this application, Policy manager concludes that network is actually needed every 12.5 minutes (on average). PM now adjusts a policy for this application, disables the network interface and schedules it to be turned on every 12.5 minutes.

3.4 Power Management Controller

The main function of the power management controller is to execute power management polices by controlling the states of hardware devices. From architecture point of view, the system is a set of atomic components, which interact within the system, some of them are power-managed externally. The activity of the components is coordinated by the Operating system's kernel. The operating system provides a hardware abstraction API, which device's driver need to implement. The kernel communicates with the devices through the device's drivers. Operating System provides a special ACPI abstraction for power management. Power manageable components have a set of ACPI drivers responsible mainly for management of the operation states. As shown in figure 3.3, ACPI Layer is next to drivers layer, between the kernel and the hardware. Which suggests that devices coordinated by regular drivers, and power managed commands are delegated to ACPI drivers. Linux OS provide an interface as mentioned previously which device's driver need to implement in order to be power managed.

In our framework we rely on operating system's ACPI layer, and all power management control is executed though a standard API.

Another responsibility of the PMC is to maintain the time based policies. Once established by the Policy manager, the policy needs to be scheduled (e.g switch network on every 5

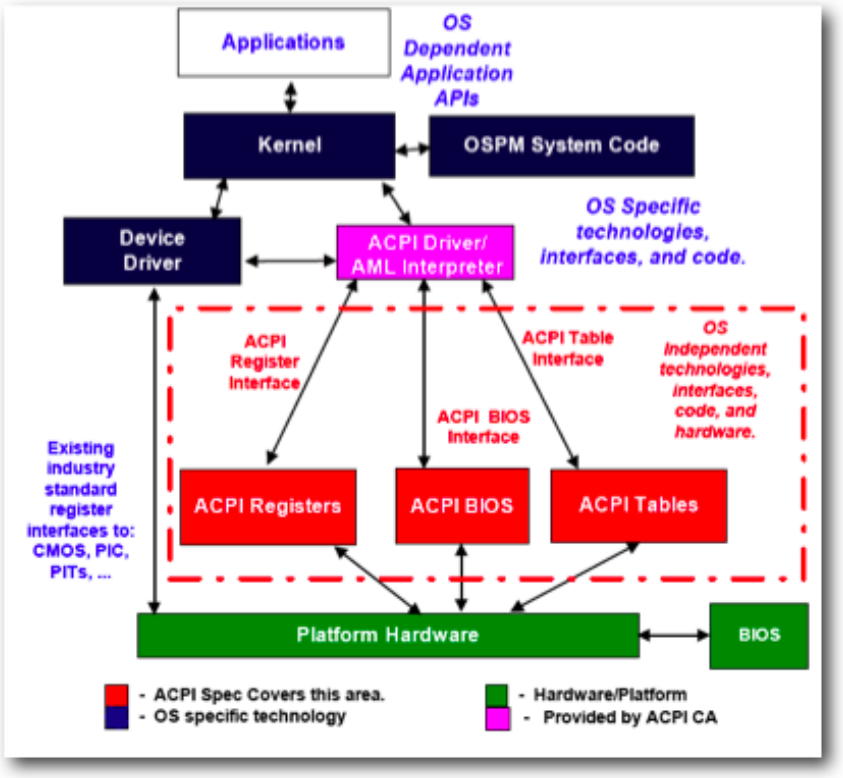


Figure 3.3: OSPM/ACPI Global System [16]

minutes for 2 minutes period). The mechanism of scheduling and de-scheduling change events is a function of the Power Management Controller.

3.5 Summary

The Power Management Framework in PowerRunner relies on Operating System abstraction for power management, although we implemented our solution on Android OS which implement ACPI spec., its not bound to a particular OS as long as it provides an API for power management of hardware . The uniqueness of our implementation lies in the intelligent power policy manager, which take into account User's activity and applies a range of DPM schemes to maximise energy economy. The modular structure of the observer module designed for lightweight and non intrusive instrumentation of the system.
In the next chapters, we describe in detail the estimation power model, power management techniques and the concrete implementation of the software.

Chapter 4

Power Model

We now discuss our approach for modelling the estimation of consumption of Power Manageable Components (PMC).
As we explained in Section 2.3, PMC's usually have several operating modes which has power cost, and transition delay. Knowing the power consumption of individual components in the system, allow to estimate the total power consumption of an application. Thus an application has a cost in power units (mW). Since we know the capacity of the battery, we can estimate the power budget of an application. In other words the power estimation model allow to predict the battery lifetime and under a workload of a particular application. The model is also enable to forecast power budget requirements for activities in the future.
Estimating power consumption in real time, is not a trivial task considering we don't know the current flowing into the system from the battery (It's not reported by the OS, when device is running of the battery), and even if we would know, it wouldn't be sufficient for estimating the consumption of each of the components. In [22], Bircher et al. proposed a data mining technique for estimation consumption of system components, from total energy consumed by the system.
To build an power estimation model, we used linear regression analysis. For linear regression we modelled a mobile system as a set of variable corresponding to operation states of hardware components. We implemented an estimation model by collecting samples of hardware component's states, and current flowing into the system, then we used a linear regression to extract the coefficients of power consumption for a set of variables which represents different operation modes and workloads.

4.1 Environment Setup

The target mobile device in our project is HTC G1, a phone developed by HTC, which runs Android OS developed by Google. Although our implementation is on a specific device, our contributions could easily be replicated on other platforms.

4.1.1 Android Architecture

Android is built on top of a solid and proven foundation: the Linux Kernel. Linux provides the hardware abstraction layer for Android, allowing Android to be ported to a wide variety of hardware platforms.

The Android platform uses Linux kernel for its memory management , process management, networking, power management and other operating systems services

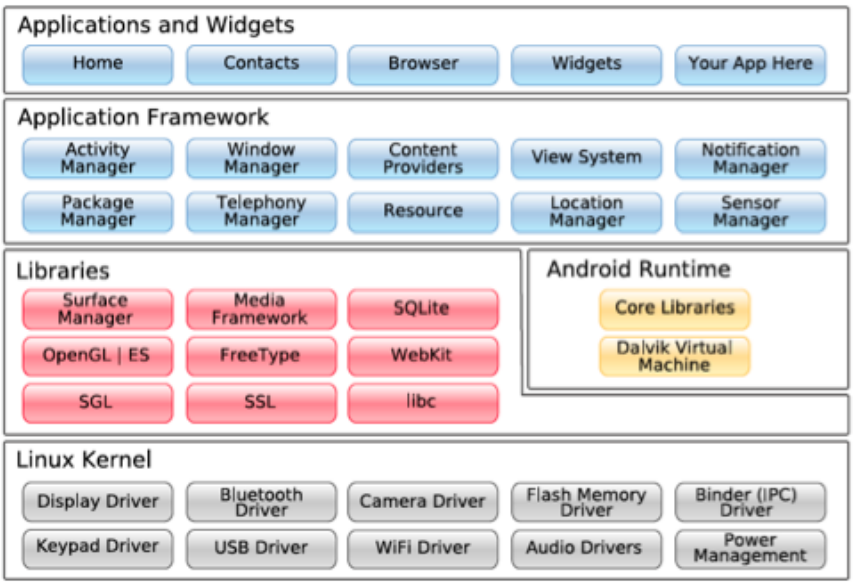


Figure 4.1: Android System Architecture [23]

As shown in figure 4.1, the next layer above the kernel is the layer of Androids native libraries. These shared libraries are written in C or C++, compiled for specific device the Android is running on, and preinstalled by the device's vendor.

Some of most important native libraries include the following:

- Surface manager

- 2D and 3D graphics
- Media Codecs
- SQL Database engine
- Browser engine

Next to native libraries, above the kernel sits the Android RunTime layer, which contains the Dalvik virtual machine and the native Java libraries. Dalvik VM is Googles implementation of Java , optimised for mobile devices. All the code written for android is in Java language and executed in Dalvik VM. The layer above is the Application Framework. This layer provides the high level building blocks , used for application development. This framework is pre-installed on Android Os and can be extended with custom components as needed. Applications and Widgets Layer is the top of the iceberg, end users can see only those programs on their devices, without knowing all the action going inside the system.

4.1.2 Hardware Architecture

In our project we used Android Developer Phone 1, a rooted[1] version of HTC G1. We use the Android 1.6 Operating System, and same version of SDK. The Android platform operate on modified version of Linux kernel 2.6.25.
A high level of HTC Dream architecture is shown in figure 4.2

The G1 consist of the following components:

- Qualcomm MSM7201A chipset
- Sharp 3.2-inch TFT-LCD flat touch-sensitive screen with 320 x 480 (HVGA) resolution
- 1150 mAh lithium-ion battery
- QDSP4000 and QDSP5000 high-performance digital signal processors
- Quadband GPRS and EDGE network interfaces
- Bluetooth chip
- Wifi network interface
- ARM 11 processor
- ARM 9 modem processor

[1]Some of our optimisations requires root access, which we'll discuss in Chapter 5.

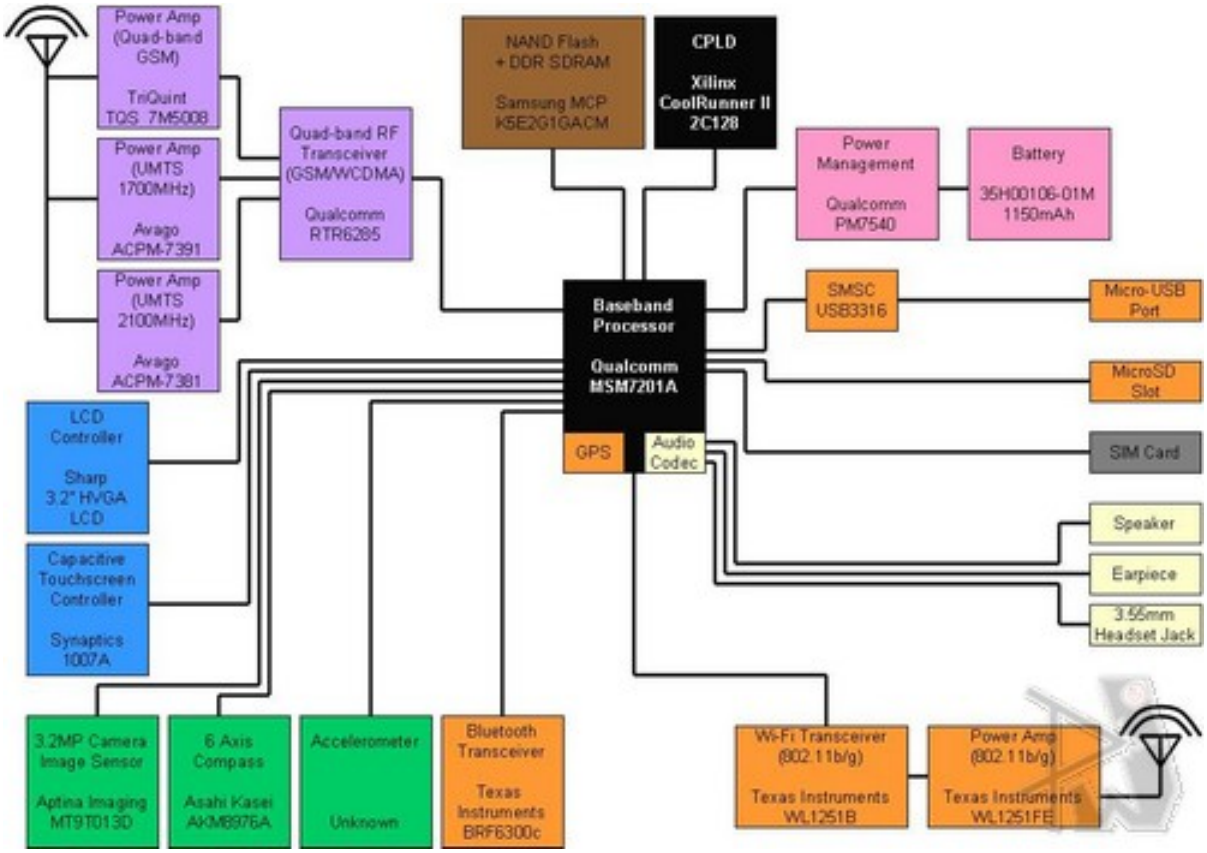
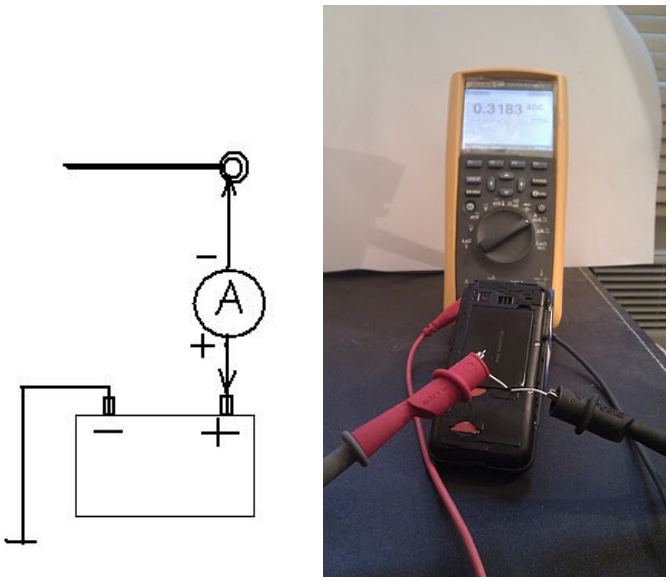


Figure 4.2: HTC Dream hardware architecture [24]

In G1, ARM 11 processor is operating as central processing unit (CPU), it runs the Android OS and execute the applications on the device. According to ARM specification it support DFS (Dynamic frequency scaling), and can operate at 124Mhz, 246Mhz, 384Mhz and 528Mhz [25]. In practice it operates only on 246Mhz and 384Mhz, because at 124Mhz the system is unstable and 528Mhz is not used.



(a) Instrumentation setup (b) Fluke 289 instrumenting HTC G1

Figure 4.3: Real current measurement

4.1.3 Current Measurement Setup

We constructed the power estimation model, using real power measurements. We probed the contact between the battery and the phone and measured the current using Fluke 289 Multimeter. In order to measure the contact, we had to isolate the '+' contact between the battery and the device with power isolation band, and connect wires for each (override the connection). Alligator clumps from the multimeter connected to each of the wires. In other words the current flows through the multimeter, is it became part of the circuit 4.3a.

We used wires with low resistance (extracted from SATA cable), to get measurements as accurate as possible [26] :

- Current rating : 1 Amp
- Insulation Resistance : $< 1000M\Omega$
- Contact Resistance : $< 30m\Omega (< 45m\Omega \; after \; stress)$.

To calculate the power, we use battery voltage figures reported by the Operating System. Figure 4.3b shows the setup of instrumentation.

4.2 Building Power Model

We now explain our approach for power estimation model on mobile device, which hardware components and operation modes we took into account. As well as how we collected and consolidated the power probes with system states. And the final results we found and used in our Power Management Framework.
In our model the mobile system as a whole operates in two distinct states "StandBy" and "Active". In StandBy state the device is in low-power sleep mode, all hardware components including the CPU are idle, except the modem processor. The power consumed in StandBy state is almost constant (measured to be around 80mW). The system is in Active state when CPU is operational. The system is in Active state when the screen is on, or if a wake lock is held to insure the CPU is running. In contrast to Idle state, the power consumed in Active state is considerably higher ($250mW \approx 2000mW$), and significantly affected by the workload. Our estimation model focuses on modelling energy consumption of hardware components in Active state, as we assume consumption in Idle state is constant. We consolidated the operating modes of a set of hardware devices with current measurements, and used Linear Regression method to build the model. We used Linear regression for fitting the dependant variable (current flowing into the system, in mA) to a set of independent variables (hardware modes) by corresponding linear coefficients.

4.2.1 Choosing Variables

Since each device has its own characteristics of operation, which affect the power consumption at any point of time. There is a relationship between the total power consumption of the system and the operation modes of hardware components. To find this relationship as a linear dependancy between the current and the operating modes we used a linear regression method. In the linear regression model we defined the current (in mA) as a scalar variable (dependant variable). The operating modes of hardware components we defined as independent variables in the model.

We modelled majority of the hardware components on HTC G1 as listed below (values they can take in brackets):

- **CPU:** The CPU refers to ARM11 processor, which is used to run the operating system and user application. In HTC G1 device it operates in two frequency modes (246Mhz and 384Mhz), therefore we defined two variables which capture the operating frequency and utilisation. CPU variables ranges 0-100 (Utilisation %)
  - CPU_Full_util (0-100)
  - CPU_Med_util (0-100)

- **Wifi:** The Wifi network interface, we modelled as three variables, to capture when its ON, is there network traffic, and the packet rate. Thus we defined the followings parameters:
  - Wifi_ON (0,1)
  - Wifi_Traffic (0,1)
  - Wifi_Packet_Rate (≥ 0)
- **LCD Screen:** The screen variables capture, whether the screen is on, and the brightness reported by the OS (ranges in value from 0 to 255):
  - Screen_State (0,1)
  - Screen_Brightness (0-255)
- **GPS:** GPS have four states, OFF, TURING_ON, ON, TURNING_OFF. From power consumption point of view, the last three can be viewed as ON (there is no significant overhead, while GPS is fixing on satellites). Therefore we defined one variable capturing the state (ON/OFF) of GPS component: GPS_ON (0,1).
- **Network:** We modelled the modem network activity similarly to WIFI interface, and defined three variables :
  - Network_ON (0,1)
  - Network_Traffic (0,1)
  - Network_Packet_Rate (≥ 0)
- **Audio:** Android SDK provides a convenient API for sound device driver, which allow to query where there audio stream is playing, and is the speaker turned on. Thus we defined two variables:
  - Audio_On (0,1)
  - Speakerphone_On (0,1)
- **Telephony:** Android provides an API for Telephony Layer of the platform. We considered two states of the modem, when the phone is ringing, and when the call is off-hook. Thus we defined two parameters:
  - CALL_STATE_OFFHOOK (0,1)
  - CALL_STATE_RINGING (0,1)
- **SD Card:** We captured the state of the SD card, when there is read/write activity, thus we defined:

- SD_ACTIVITY (0,1)

- **MISC:** The power consumption, not accounted with variables listed above, is captured in a single variable MISC. It is nothing but the constant y-intercept in a linear regression model, which will be described in the following Section 4.2.5

4.2.2 Current Measurement

We built the power estimation model using real current measurements. We instrumented the the plus contact between the phone and the battery, with Fluke 289 Multimeter. Fluke289 is a highly accurate device, that capable of measuring current as low as $500\mu A$, and record the measurements, which later can be exported to a PC. The idea is to correlate the current measurement with system state (a collection of parameters).
The information is gathered from two different sources: one from the logger application running on the device, and the other from multimeter. The correlation was possible only by timestamps of the samples of both sources. The challenge was the time synchronisation between the two. The FlukeView software, which we used to export the measurements from the multimeter device, has an option to offset the timestamps of samples to local clock (where FlukeView is running). Thus we had to synchronise the clocks between the mobile device and the PC, before every measurement session. We used NTP protocol to achieve this goal, PC was a NTP server (synced with external Stratum 2 server), while mobile was a client. G1 synchronised with PC over the wireless network. In fact Fluke 289 measure the current flow at rate of 1Khz, while its recording an interval of ≈ 1 second for every sample. Each sample Multimeter records have a measurement of minimum, maximum and average current within an interval. Although a delay between the mobile and PC was in range of few microseconds, it didn't affect the accuracy of the samples. For every system-state sample we took on G1, we took an average current measurement for an interval of 1 second.

In figures 4.4 and 4.5 shown a current measurement session output.

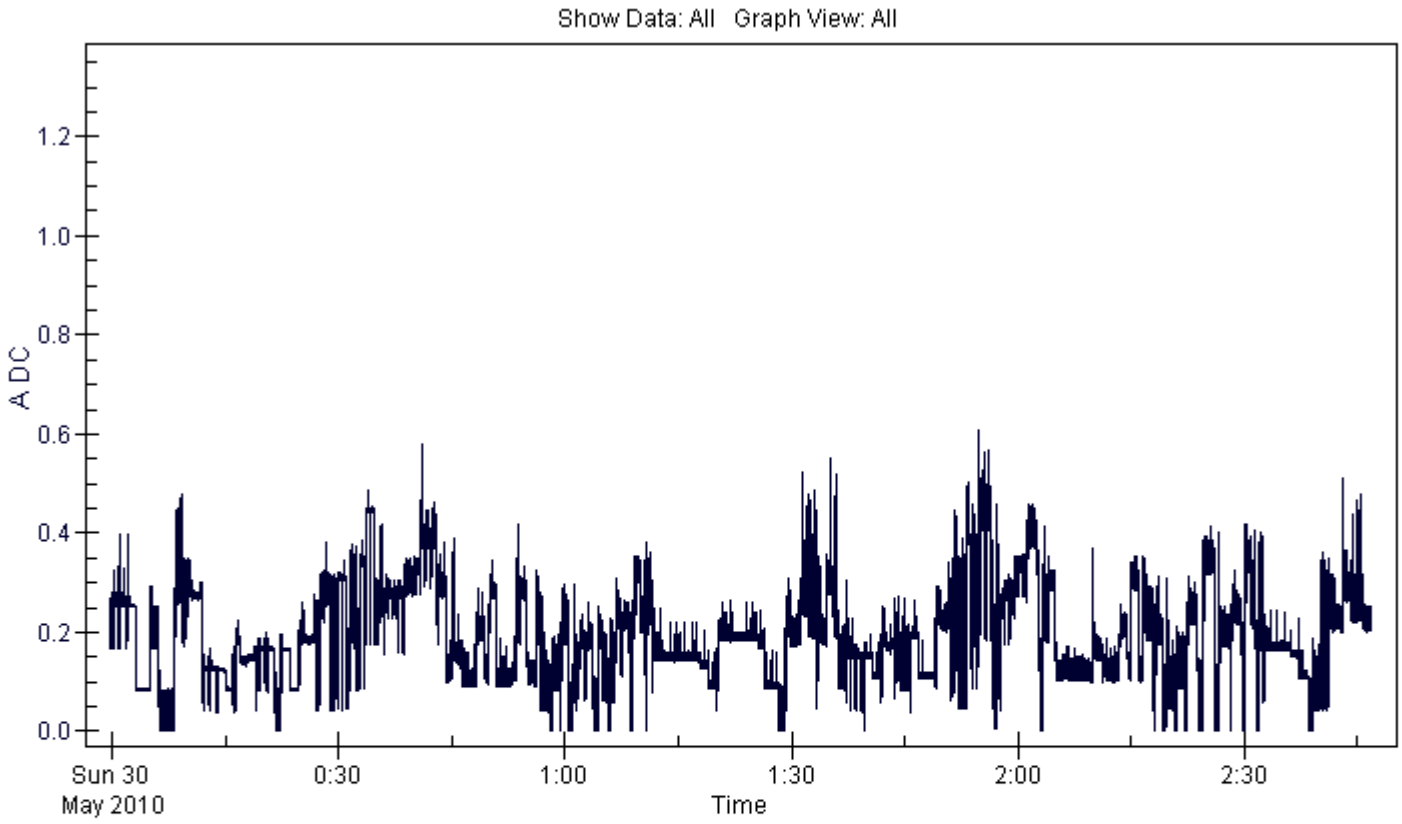


Figure 4.4: Current measurement recording

FLUKE 289	V1.10	13060158								
Start Time	Stop Time	Elapsed Time	Interval	Total readings	Intervals	Input Events	Session Name			
##########	##########	02:46:59	00:00:01	15051	10021	10061	Save 1			
Max Time	Max	Average	Min	Min Time	Scaling					
##########	0.6086 A DC	0.1898 A DC	0.0012 A DC	##########	(none)					
To display full timestamps use Format Cells 'm/d/y hh:mm:ss.0'										
Reading	Sample	Start Time	Duration	Max Time	Max	Average	Min Time	Min	Description	Stop Time
1	0.1877 A DC	59:47.9	00:00.4	59:48.1	0.1931 A DC	0.1817 A DC	59:48.2	0.1701 A DC	Interval	59:48.3
2	0.1971 A DC	59:48.3	00:01.0	59:49.2	0.2031 A DC	0.1846 A DC	59:48.7	0.1716 A DC	Interval	59:49.3
3	0.1702 A DC	59:49.3	00:00.2	59:49.4	0.2079 A DC	0.1891 A DC	59:49.3	0.1702 A DC	Unstable	59:49.5
4	0.1776 A DC	59:49.5	00:00.8	59:49.7	0.2008 A DC	0.1812 A DC	59:50.3	0.1695 A DC	Interval	59:50.4
5	0.1873 A DC	59:50.4	00:00.9	59:50.6	0.2023 A DC	0.1843 A DC	59:50.7	0.1705 A DC	Unstable	59:51.3
6	0.2154 A DC	59:51.3	00:00.1	59:51.3	0.2154 A DC	0.2154 A DC	59:51.3	0.2154 A DC	Interval	59:51.4
7	0.1707 A DC	59:51.4	00:00.3	59:51.5	0.1931 A DC	0.1838 A DC	59:51.4	0.1707 A DC	Unstable	59:51.7
8	0.2582 A DC	59:51.7	00:00.7	59:52.0	0.2666 A DC	0.2601 A DC	59:52.3	0.2556 A DC	Interval	59:52.4
9	0.1859 A DC	59:52.4	00:00.9	59:53.0	0.2053 A DC	0.1839 A DC	59:52.7	0.1696 A DC	Unstable	59:53.3
10	0.2202 A DC	59:53.3	00:00.1	59:53.3	0.2202 A DC	0.2202 A DC	59:53.3	0.2202 A DC	Interval	59:53.4
11	0.1771 A DC	59:53.4	00:00.2	59:53.5	0.2461 A DC	0.2116 A DC	59:53.4	0.1771 A DC	Unstable	59:53.6
12	0.2675 A DC	59:53.6	00:00.7	59:53.6	0.2675 A DC	0.2609 A DC	59:54.0	0.2571 A DC	Interval	59:54.3
13	0.2620 A DC	59:54.3	00:00.8	59:54.3	0.2620 A DC	0.2438 A DC	59:55.0	0.2077 A DC	Unstable	59:55.1
14	0.1764 A DC	59:55.1	00:00.2	59:55.2	0.1931 A DC	0.1848 A DC	59:55.1	0.1764 A DC	Interval	59:55.3
15	0.2347 A DC	59:55.3	00:00.5	59:55.3	0.2347 A DC	0.2005 A DC	59:55.5	0.1796 A DC	Unstable	59:55.8
16	0.2551 A DC	59:55.8	00:00.5	59:56.2	0.2700 A DC	0.2638 A DC	59:55.8	0.2551 A DC	Interval	59:56.3
17	0.2647 A DC	59:56.3	00:00.9	59:56.5	0.2665 A DC	0.2183 A DC	59:56.8	0.1718 A DC	Unstable	59:57.2
18	0.2530 A DC	59:57.2	00:00.1	59:57.2	0.2530 A DC	0.2530 A DC	59:57.2	0.2530 A DC	Interval	59:57.3
19	0.2603 A DC	59:57.3	00:01.0	59:58.0	0.2726 A DC	0.2622 A DC	59:58.2	0.2541 A DC	Interval	59:58.3

Figure 4.5: Fluke 289 output in CSV format (truncated).

4.2.3 Logging Session

Once we choose the parameters for our power estimation model, the next step was to collect the samples for the linear regression model. We did analysis of linear dependency between system's power consumption and hardware utilisation. In other words: we found what is the power consumption of each of the individual components. Thus we gathered and combined samples, which represent a state of the system in term parameters defined above, and the power measurements. We used the Observer module within our architecture that monitors the states of hardware components on the device. We implemented a logger utility class for it, which takes a system state snapshot, and prints out a set of comma separated values into a log file, with a timestamp for each of the samples. An example of a samples is shown in table 4.1:

In total we collected around 10,000 samples during several hours, and several sessions. In every session we ran a series of load test which put load on individual hardware components, or combination of them, in order to make samples noise free as possible. Some of the tests included CPU stress, Network speed test, SDcard test, Music player and so on. In each test we changed the screen brightness to capture power consumption of the screen. During each logging session on the device, the multimeter was instrumenting and recording the real current flowing into the system. The procedure of merging two logs will be explained in the following section 4.2.4

Table 4.1: System state samples

1	30/05/2010 00:00:03	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
2	30/05/2010 00:00:04	0	1	0	68	68	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
3	30/05/2010 00:00:05	0	1	0	8	8	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4099
4	30/05/2010 00:00:06	0	1	0	43.56	43.56	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
5	30/05/2010 00:00:07	0	1	0	23	23	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
6	30/05/2010 00:00:08	0	1	0	19	19	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4096
7	30/05/2010 00:00:09	0	1	0	9.901	9.901	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4096
8	30/05/2010 00:00:10	0	1	0	27	27	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4100
9	30/05/2010 00:00:11	0	1	0	6	6	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4100
10	30/05/2010 00:00:12	0	1	0	28.71	28.71	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
11	30/05/2010 00:00:13	0	1	0	9	9	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
11	30/05/2010 00:00:13	0	1	0	9	9	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4096
13	30/05/2010 00:00:16	0	1	0	98.17	98.17	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4086
14	30/05/2010 00:00:17	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4097
15	30/05/2010 00:00:18	0	1	0	35.35	35.35	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	337	4097
16	30/05/2010 00:00:19	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4086
16	30/05/2010 00:00:19	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4086
17	30/05/2010 00:00:20	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4086
18	30/05/2010 00:00:21	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4083
19	30/05/2010 00:00:22	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4086
20	30/05/2010 00:00:23	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4082
21	30/05/2010 00:00:24	0	1	0	100	100	0	0	0	1	255	0	0	0	0	0	0	0	0	0	0	0.00	0	1	333	4082

4.2.4 Consolidation and Analysis

We used IBM SPSS Statistics software to run regression analysis. The software expected a spreadsheet with samples. As we used current measurements (mA) as a dependant variable in our linear regression model, we had to merge both logs into one. Assuming the clocks were synchronised we could correlate, every device sample to a multimeter sample by a corresponding timestamp. As there were large amount of samples, doing this work by hand was time consuming and error prone, we automated the procedure by loading both datasets into a MYSQL database and running an OLAP query to join the two by timestamp. The result of a join query, we exported into a CSV file, which we used in statistics software to build the linear regression model.

4.2.5 Linear Regression Model

The linear regression approach for estimating power consumption, results in highly accurate model, which allows not just estimate consumption of the whole system, but a consumption of each individual hardware devices. Consider a measurement of device j (e.g CPU utilisation) in probe i is $\theta_{i,j}$, power consumed by the corresponding device $\pi_{i,j}$ for the parameter coefficient μ_j is:

$$\pi_{i,j} = \theta_{i,j} \cdot \mu_j \tag{4.1}$$

The power consumption not captured by any of the parameters aggregated into a constant offset λ. Thus the total power consumption for a probe i with n measurement parameters

can be represented as a sum of corresponding n values and additional constant offset λ:

$$\Pi_i = (\pi_{i,0} + \pi_{i,1} + \pi_{i,2} + \cdots + \pi_{i,n}) + \lambda \tag{4.2}$$

Substitute 4.1 into 4.2 :

$$\Pi_i = ((\theta_{i,0} \cdot \mu_0) + (\theta_{i,1} \cdot \mu_1) + (\theta_{i,2} \cdot \mu_2) + \cdots + (\theta_{i,n} \cdot \mu_n)) + \lambda \tag{4.3}$$

For each probe i, let $x_i = (\theta_{i,0}, \theta_{i,1}, \theta_{i,2}, \ldots, \theta_{i,n})$, and $\mu = (\mu_0, \mu_1, \mu_2, \ldots, \mu_n)$

We can reduce 4.3 to :

$$\Pi_i = x_i \cdot \mu + \lambda \tag{4.4}$$

Once we establish the constant offset λ and parameter coefficients for hardware devices $\vec{\mu}$, for every probe of the system x_i we can estimate:

- System's total consumption (at the time of the sample) : use set of measurements x_i and equation 4.4
- Hardware component consumption : use measurement parameter for corresponding component and equation 4.1. In case state of individual device measured by several parameters, to estimate power, corresponding values need to be summed.

For m samples , this model can be represented in form of:

$$\begin{pmatrix} \Pi_0 \\ \Pi_1 \\ \vdots \\ \Pi_m \end{pmatrix} = \begin{bmatrix} \theta_{0,0} & \theta_{0,1} & \cdots & \theta_{0,n} \\ \theta_{1,0} & \theta_{1,1} & \cdots & \theta_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{m,0} & \theta_{m,1} & \cdots & \theta_{m,n} \end{bmatrix} \begin{pmatrix} \mu_0 \\ \mu_1 \\ \vdots \\ \mu_m \end{pmatrix} + \lambda \cdot \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{4.5}$$

Letting:

$$\Pi = \begin{pmatrix} \Pi_0 \\ \Pi_1 \\ \vdots \\ \Pi_m \end{pmatrix}, X = \begin{bmatrix} \theta_{0,0} & \theta_{0,1} & \cdots & \theta_{0,n} \\ \theta_{1,0} & \theta_{1,1} & \cdots & \theta_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{m,0} & \theta_{m,1} & \cdots & \theta_{m,n} \end{bmatrix}, \epsilon = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

The model can be represented in form of:

$$\Pi = X\mu + \lambda \cdot \epsilon \tag{4.6}$$

Total energy E consumed by the system across a set of probes x_i in interval of t_p samples, may be estimated by a sum:

$$E = \sum_{i=0}^{m} t_p \cdot \Pi_i = t_p \sum_{i=0}^{m} (x_i \cdot \mu + \lambda) \tag{4.7}$$

The power consumed by the system in Active state, is estimated by the linear regression model. In the linear regression model for the system in Active state λ represents the coefficient for variable **MISC**

4.3 Results and Validation

Table 4.2 presents the hardware variables and the corresponding coefficients in the power estimation model.

Table 4.2: Power Estimation Model

Hardware Component	Variable	Description	Value of $\theta_{i,j}$	Coefficient μ_j	Units
CPU	**CPU_Full_util**	**CPU Utilisation @ 384Mhz**	**0-100**	**47.30**	**mA / %**
	CPU_Med_util	**CPU Utilisation @ 246Mhz**	**0-100**	**29.83**	**mA / %**
WIFI	**Wifi_ON**	**Interval of time where Wifi is ON**	**0,1**	**15.61**	**mA**
	Wifi_Traffic	**Interval of time where there is Wifi traffic**	**0,1**	**69.44**	**mA**
	Wifi_Packet_Rate	**Packet transfer rate in interval of time**	≥ 0	**0.36**	**mA / packet**
LCD Screen	**Screen_State**	**Interval of time where screen is ON**	**0,1**	**46.40**	**mA**
	Screen_Brightness	**Screen Brightness**	**0-255**	**0.56**	**mA/step**
GPS	**GPS_ON**	**Interval of time where GPS is ON**	**0,1**	**75.75**	**mA**
Telephony Data Network	**Network_EDGE**	**Interval of time where Network is ON**	**0,1**	**89.79**	**mA**
	Network_Traffic	**Interval of time where there is Network traffic**	**0,1**	**50.45**	**mA**
	Network_Packet_Rate	**Packet transfer rate in interval of time**	≥ 0	**0.02**	**mA / packet**
Audio	**Audio_On**	**Interval of time where Audio is ON**	**0,1**	**48.56**	**mA**
	Speakerphone_On	**Interval of time where Speaker is ON**	**0,1**	**23.37**	**mA**
Telephony	**CALL_STATE_OFFHOOK**	**Interval of time where there is a phone call**	**0,1**	**155.96**	**mA**
	CALL_STATE_RINGING	**Interval of time where phone is ringing**	**0,1**	**178.87**	**mA**
SD Card	**SD_Activity**	**Interval of time where there activity on the SD card**	**0,1**	**1.89**	**mA**
Miscellaneous	**MISC**	**Interval of time system is Active**	**1**	**45.18**	**mA**

To validate the accuracy of the model we collected additional set of samples. We used this set to approximate the error rate of our power estimation model. Equation 4.4 provides a current estimate for each sample i. The error approximation is the percent of absolute relative error :

$$error = 100 \cdot |\frac{actual - estimated}{actual}|\% \tag{4.8}$$

The results suggest that the power estimation model accurately predicts the power consumption of the system. Over all the samples we achieved $< 0.4\%$ mean error. The median error across all the samples is 8%

4.3.1 Summary

We built a power estimation model, based on linear regression analysis which uses system low level measurements of hardware components to estimate power consumption of mobile device system. We showed that the model is highly accurate, and that it can be used to analyse the power consumption breakdown by individual hardware components in a mobile device. By analysing the coefficients of linear regression parameters we identified hardware components which are more expensive in terms of power cost. And applying appropriate optimisations on those components we can save a significant amount of energy. One of the benefits of the estimation model we presented, is the power consumption analysis in run-time. For testing framework (Chapter 6.3), it is fundamental requirement to be able to analyse system power consumption in runtime, under workload of a tested software application. In the following chapters we present the implementation of the PowerRunner system, discuss optimisation techniques, and overview the application of the system as a testing framework for developers.

Chapter 5

Implementation

We now present the implementation of PowerRunner software application. In this project we implemented PowerRunner on Android OS. The infrastructure we have used in the scope of this project is provided by Android platform. Although features like SQLITE abstraction, Multithreading, Inner Process Communication (Intent), Background Services and more, are specific to Android, conceptually its not bound, and easily can be replicated on other platforms. As we discussed in Chapter 3, we have three main modules in the system: The Observer, The Controller and The Policy Manager. We now discuss implementation of each of the modules and the interaction between them.

5.1 The Observer

In chapter 3, we discussed the function of the observer module in power management framework. The main responsibility of Observer is to collect information from the system about hardware components and applications. This information is processed and analysed by the policy manager. In this section we introduce our implementation of the observer module. To illustrate the challenges, we want remind reader that mobile architecture is by its nature limited in resources. Therefore one of the fundamental requirements in our implementation, was to make it lightweight and introduce as low overhead to the system as possible.

At the high level, observer can be seen as a manager for a group of monitors. Each hardware component has a dedicated monitor. Observer's function is to coordinate the monitor‘s activity and collect data from the monitors synchronously. Then Observer writes the state of the whole system into a log file, and passes it to the policy manager.

In Android exists a concept of 'Service'. It is providing two main features [27]:

- A facility for the application to tell the system about something it wants to be doing in the background (even when the user is not directly interacting with the application).
- A facility for an application to expose some of its functionality to other applications. This corresponds to calls to Context.bindService(), which allows a long-standing connection to be made to the service in order to interact with it.

We implemented observer as a dedicated thread, in a context of background service. The rationale behind this was to be able to collect system information, while user can run other applications.

Observer spawns monitor threads for each of the devices, we have now details from each of the monitors.

5.1.1 Hardware Monitoring

In this project we implemented a set of monitors for hardware devices. As we discussed earlier Observer thread acts as coordinator for the monitors. Monitors are also implemented as threads. Each monitor thread, collects information about hardware it's responsible for and stores it as a MonitorProbe object in the local BlockingQueue. Each Monitor has it's own implementation of MonitorProbe.

We now discuss the specifics of each of the hardware devices. How the data is harvested, and which part of the information about hardware is relevant to our Power Management Framework.

5.1.2 Monitor probe interface

We have put an infrastructure in place, for transforming probed information into power units. For example: a screen monitor samples the system and collects a sample for some interval of time. This probe is now holding a state of the screen for a specific interval of one second. By using of the power estimation model, and the voltage reported, we can translate a screen state into power units. State saying screen is on , brightness is 135 units and voltage of 4.2V would be equivalent to:

$$(46.677 + 135 \times 0.557) \times 4.2 = 511.82mW \tag{5.1}$$

The infrastructure is an interface for a probe:

****
public interface MonitorProbe

public double getPower();
public void addPower(long batt_voltage);
public String getType();

****

5.1.3 CPU Monitor

This monitor collects usage statistics of the CPU, and calculates the utilisation for a fraction of sample time :

- frequency
- systemUsage
- userUsage
- userSystemUsage

To get the frequency value we parse the system file "/proc/cpuinfo" and extract BogoMIPS value. A typical contents of the cpuinfo file as following:

cat /proc/cpuinfo
Processor : ARMv6-compatible processor rev 2 (v6l)
BogoMIPS : 383.38
Features : swp half thumb fastmult edsp java
CPU implementer : 0x41
CPU architecture: 6TEJ
CPU variant : 0x1
CPU part : 0xb36
CPU revision : 2

Hardware : trout
Revision : 0080
Serial : 0000000000000000

Usage statistics is harvested from a system file ''/proc/stat''. stat file hold useful statistics about the cpu usage, we can see typical contents bellow:

cat /proc/stat
cpu 5846 604 2508 30881 246 1 1 0 0
cpu0 5846 604 2508 30881 246 1 1 0 0
intr 215654 1825 0 0 0 0 3531 0 19613 0 0 0 0 0 0 0 2672 0 0 1986 343 103892 0 29 0 0 875 1 0 0 0 0 0 74654 0 0 0 0 0 0 0 0 0 0 0 0 2941 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 2 0 0 1242 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1631 0 0 0 0 0 412 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ctxt 295515
btime 1276432844
processes 461
procs_running 1
procs_blocked 0

The very first ''cpu'' line aggregates the numbers in all of the other ''cpuN'' lines. These numbers identify the amount of time the CPU has spent performing different kinds of work. Time units are in USER_HZ or Jiffies (typically hundredths of a second). The meanings of the columns are as follows, from left to right [1] :

1. user: normal processes executing in user mode
2. nice: niced processes executing in user mode
3. system: processes executing in kernel mode
4. idle: twiddling thumbs
5. iowait: waiting for I/O to complete
6. irq: servicing interrupts
7. softirq: servicing softirqs

The total CPU time is a sum of all CPU times above. The CPU usage on user's tasks; is the proportional time CPU spent on user's processes (normal and niced) to the total time. The percent of CPU utilisation on user's and systems tasks is the proportion of user's processes and system's processes to the total CPU time.

CPU monitor uses the following formulas, to calculate the total time, user and system usages :

$$totalCpuTime = user + nice + system + idle + iowait + irq + softirq \tag{5.2}$$

$$userUsage = \frac{user + nice}{totalCpuTime} \tag{5.3}$$

$$userSystemUsage = \frac{user + nice + system + irq + softirq}{totalCpuTime} \tag{5.4}$$

For every CPU probe we want to calculate the CPU utilisation, for an interval of time. To achieve this, the monitor keeps in memory previous probe reading, and usage calculations. For every probe we find the deltas for user and system usages and totalCpu time, between the current and previous probes. Then we use deltas to calculate utilisation as a proportion of usage to total CPU time. Thus:

$$userUsage = \frac{deltaUserTime}{deltaTotalTime} \times 100(\%) \tag{5.5}$$

[1] According to LinuxHowTos website: http://www.linuxhowtos.org/System/procstat.htm [28]

5.1.4 Screen Monitor

Screen Monitor‘s function is to sample the state of the screen and the backlight brightness.

The later we read directly from a system file */sys/class/leds/lcd-backlight/brightness*. The former is little more complicated. Unfortunately in this version of SDK, there is no API to query the state of the screen (this feature introduced in Android 2.0). So, we took different approach to overcome this absence of API. Screen monitor starts with (screen_on=true) and listens for notification messages : *ACTION_SCREEN_OFF* and *ACTION_SCREEN_ON*, which system broadcasts when corresponding even occurs. The same listener, notifies policy manager about changes in the state on the screen (some optimisations [Section 5.3], based on fact that screen is off).

Screen monitor stores the state of the screen in *LCDProbeInfo* object, and holds the following:

- lcdOd [boolean]
- lcdBrightness [int]

5.1.5 Audio Monitor

Android SDK provides an abstraction for audio drivers, and enables us to query in runtime the state of the audio component. Knowing what the state of the audio is, we can try to analyse user's context (listening to music, talking to someone through speakerphone, etc...). Audio monitor probe contains the following information:

- musicActive [boolean]
- bluetoothHeadsetOn [boolean]
- speakerphoneOn [boolean]

5.1.6 Battery Monitor

We need to monitor the battery to know useful information about the energy resources on the phone. This is information like voltage, charging state, capacity of the battery and so on. Luckily Linux stores state of the battery is system files:

/sys/class/power_supply/battery/status;
/sys/class/power_supply/battery/health;
/sys/class/power_supply/battery/present;
/sys/class/power_supply/battery/technology;
/sys/class/power_supply/battery/capacity;
/sys/class/power_supply/battery/batt_id;
/sys/class/power_supply/battery/batt_vol;
/sys/class/power_supply/battery/batt_temp;
/sys/class/power_supply/battery/batt_current [2];
/sys/class/power_supply/battery/charging_source;
/sys/class/power_supply/battery/charging_enabled;
/sys/class/power_supply/battery/full_bat;

The monitor periodically reads those files, and stores the values in BatteryProbeInfo object, which persist in local BlockingQueue.

5.1.7 Bluetooth Monitor

The Bluetooth Monitor, probes the state of bluetooth device. In Android 1.6 there is no public API available to query the state of bluetooth device. For this monitor, we found a workaround by subscribing to an internal system topic, and listen to BlueTooth state change events.

System intent: *android.bluetooth.intent.BLUETOOTH_STATE*

The monitor initialised with a value we retrieve from system settings *Settings.Secure.BLUETOOTH_ON*. Ones the monitor initialised, it reports to observer the current state of bluetooth device, until receiving a notification and change of state.

Bluetooth device have four power states [27] :

- int BLUETOOTH_STATE_OFF = 0;
- int BLUETOOTH_STATE_TURNING_ON = 1;

[2] The state of the current is reported accurately only, when the device is charging. The batt_current parameter reports 0, when the mobile device is running on battery.

- int BLUETOOTH_STATE_ON = 2;
- int BLUETOOTH_STATE_TURNING_OFF = 3;

BT monitor store in *BluetoothProbeInfo* state of bluetooth *mBluetoothState*.

5.1.8 GPS Monitor

The GPS monitor concerns with the state of GPS device. In contrast to BlueTooth, state information is available through an API. However, implementing special listener for GpsStatus. It has a callback a method *onGpsStatusChanged*, when GPS status changes event is fired. The status changes, not just between on and off states, but also when a location changes. GpsStatusListener is part of Android's LocationServices API, and can receive many kinds of notifications about the state of GPS. However we focused only on four states:

- GPS_EVENT_FIRST_FIX - fires when GPS Location is Fixed;
- GPS_EVENT_SATELLITE_STATUS - fires when GPS Status is available;
- GPS_EVENT_STARTED - fires when GPS is Started;
- GPS_EVENT_STOPPED - fires when GPS is Stopped;

To the observer Monitor pass the current state of GPS receiver.

5.1.9 Network Monitor

The network monitor responsible for monitoring the data services interface of the telephony network. The network interface has five operating modes (as discussed in 5.2.3). We aimed to capture those operation states, in our monitor. TelephonyManager API kindly provides the connection states of the device: Idle, Sleep, OFF. To capture what device is doing: transmitting or receiving we had to analyse some statistics of the network driver. In figure 5.1 we can see a typical contents of */proc/self/net/dev* file. The structure of the file split into two parts: Receive and Transmit.

```
# cat /proc/self/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
    lo:       0       0    0    0    0     0          0         0       0       0    0    0    0     0       0          0
rmnet0:   64750     241    0    0    0     0          0         0   19453     214    0    0    0     0       0          0
rmnet1:       0       0    0    0    0     0          0         0       0       0    0    0    0     0       0          0
rmnet2:       0       0    0    0    0     0          0         0       0       0    0    0    0     0       0          0
  usb0:       0       0    8    0    0     0          0         0       0       0    0    0    0     0       0          0
```

Figure 5.1: Network usage statistics

Each part contains values for :

- bytes
- packets
- errs
- drop
- fifo
- frame
- compressed
- multicast

Our Monitor calculates the transmit/receive rates, by looking at the transmitted bytes per interval of time for *rmnet0* interface. We first calculate the deltas:

$deltaTime = nowProbeTime - prevProbetime$;
$deltaTransmitBytes = currentTransmitBytes - prevTransmitBytes$;
$deltaReceiveBytes = currentReceiveBytes - prevReceiveBytes$;

Then we calculated the uplink and downlink rates as following:

$$uplinkrate = deltaTransmitBytes/1024 * 7.8125/deltaTime(bits\ per\ second) \quad (5.6)$$

$$downlinkrate = deltaReceiveBytes/1024 * 7.8125/deltaTime(bits\ per\ second) \quad (5.7)$$

Since we know the number of packets sent and received from reading the stats file, we use 5.8 to calculate the packet rate:

$$packetRate = (curPacketsReading - prevPacketsReading) * 1000/deltaTime(packetspersecond) \quad (5.8)$$

For each probe, we accurately extracts, calculate and store the following information in NetworkDataProbeInfo:

- networkDataType
- networkDataUplinkRate

- networkDataDownlinkRate
- networkDataPackateRate
- networkDataState

5.1.10 Wifi Monitor

Wifi Monitor works similarly to Network monitor, except its reading statistics for *tiwlan0* interface.

5.1.11 Phone Monitor

The purpose of Phone monitor is to observe the activity of the modem component and capture the state of the phone calls. Modem can be in three operation modes, which we capture and persist in *PhoneProbeInfo*

- CALL_STATE_IDLE
- CALL_STATE_OFFHOOK
- CALL_STATE_RINGING

The PhoneMonitor extends PhoneStateListener and overrides onCallStateChanged method to capture events of the phone. PhoneStateListener is part of Android API [27]

5.1.12 SD Card Monitor

The goal of the SD Card monitor is to capture intervals, when sdcard is busy reading or writing. For intervals when SDCard is active, monitor calculates the rate at which sectors are read or written.

Each SDProbeInfo object holds the following information:

- sectorsRead
- sectorsWrite
- sectorsTotal
- sectorsRate

The monitor analyses the disk statistics file */proc/diskstats*. In figure 5.2 shown a typical content of diskstat file.

```
# cat /proc/diskstats
   7       0 loop0 0 0 0 0 0 0 0 0 0 0 0
   7       1 loop1 0 0 0 0 0 0 0 0 0 0 0
   7       2 loop2 0 0 0 0 0 0 0 0 0 0 0
   7       3 loop3 0 0 0 0 0 0 0 0 0 0 0
   7       4 loop4 0 0 0 0 0 0 0 0 0 0 0
   7       5 loop5 0 0 0 0 0 0 0 0 0 0 0
   7       6 loop6 0 0 0 0 0 0 0 0 0 0 0
   7       7 loop7 0 0 0 0 0 0 0 0 0 0 0
  31       0 mtdblock0 0 0 0 0 0 0 0 0 0 0 0
  31       1 mtdblock1 0 0 0 0 0 0 0 0 0 0 0
  31       2 mtdblock2 0 0 0 0 0 0 0 0 0 0 0
  31       3 mtdblock3 0 0 0 0 0 0 0 0 0 0 0
  31       4 mtdblock4 0 0 0 0 0 0 0 0 0 0 0
  31       5 mtdblock5 0 0 0 0 0 0 0 0 0 0 0
  31       6 mtdblock6 0 0 0 0 0 0 0 0 0 0 0
 179       0 mmcblk0 536 5556 6155 1430 0 0 0 0 0 750 1430
 179       1 mmcblk0p1 530 5553 6083 1410 0 0 0 0 0 730 1410
```

Figure 5.2: Typical diskstat file

Each line in diskstat file, hold statistics per disk device in the following format [28]:

```
Field 1  -- # of reads issued
Field 2  -- # of reads merged
Field 3  -- # of sectors read
Field 4  -- # of milliseconds spent reading
Field 5  -- # of writes completed
Field 6  -- # of writes merged
Field 7  -- # of sectors written
Field 8  -- # of milliseconds spent writing
Field 9  -- # of I/Os currently in progress
Field 10 -- # of milliseconds spent doing I/Os
Field 11 -- weighted # of milliseconds spent doing I/Os
```

We take field 3 and 7 to analyse the number of sectors read or written in interval of time. Monitor holds in memory the previous readings. On each probe it subtracts previous from the current to get the delta of sectors in interval of time between the readings.

5.1.13 Applications Monitor

The application monitor, samples the system an collects information about running applications. Monitor extract the information from the process information pseudo-filesystem */proc*. Each process has a directory */proc/[pid]*, that contains a collection of various information files about the process. Status information about the process is held in /proc/[pid]/stat. The information includes the filename of the executable and the CPU utilisation statistics.

The monitor run trough /proc directory, and for each process subdirectory in form of */proc/[pid]* it extract: the pid, the filename and the CPU usage.

5.2 Power Controller

The main responsibility of Power Controller (PC) module is to execute state transitions of hardware components, through Operating System ACPI layer. PC retrieves change request messages from the Policy Manager (PM) and executes changes on relevant devices. For example : PM decides to switch off Wifi interface, it notifies the PC, which in turn will execute the relevant command. Because hardware components have unique characteristics, and specific power management control, we implemented a series of hardware controllers per device. In the scope of this project, we identified and implemented controllers for the most power hungry devices as candidates for optimisations :

- CPU
- Screen
- Wifi
- Modem

The hardware properties and control methods for these components will be described in this section. In Section 5.3 we will discuss the optimisation techniques for each of the devices.

5.2.1 CPU

HTC G1 uses ARM11 processor as CPU. ARM11 designed for low-power mobile devices, and have many power saving features. One of the features is the ability to switch selectively off functional units like the floating-point unit. However this feature is not controlled externally. Another important feature is the ability to slow down the clock dynamically, it operates at two frequencies: 246Mhz or 384Mhz. Also this processor has a function to

switch off the operation completely, so it consumes very little power and goes back to full operation mode when the next interrupt occurs. In general slowing down the processor clock without changing the voltage is not very effective and even might be counterproductive. Power consumed by the processor is proportional to the clock frequency,the switching capacitance of the transistor, and the square of the voltage :

$$P = CfV^2 \tag{5.9}$$

Where P is Power consumption, C is the switching capacitance of the transistor, f is the clock frequency and V is voltage.

Thus by equation 5.9, processor operating at a lower frequency consumes less power. But the time it takes the processor to complete a task is inversely proportional to the clock frequency:

$$t = 1/f \tag{5.10}$$

Where t is time takes to complete a task, f is the clock frequency of the processor.

The total energy spent on completing a task can be defined as a product of P (processor power consumption), and time t :

$$E = P \cdot t \tag{5.11}$$

From 5.11 and 5.10, we can conclude that total energy consumption is not affected by the clock frequency. Therefore, by reducing the clock frequency, computation time is increasing without reducing the energy consumed by the processor during that time. In fact, slowing down the clock might cause even higher energy consumption, as a result of the extended computation time, during which other components of the system will remain operational. Another disadvantage of DFS (dynamic frequency scaling) is a possible delay and cost of energy, while processor switching clock speeds. The only way we could save energy by slowing down CPU clock, is by reducing the voltage. However, it is not always possible and can compromise the stability of the system. On the other hand turning the processor off, is an effective method to save energy. Switching on, transition happens almost instantaneously, and has not extra power expenditure. The state of the processor remains unchanged between on/off transitions. While the processor is off not only the energy of the processor is saved, but also additional energy from other components [29].

Optimising power consumption of the CPU, can have positive effect on overall power savings. Fortunately, both schemes are implemented on Android platform, which is running on top of Linux kernel 2.6.25. On Android the processor is turns off automatically, when

screen is off for some time, unless a special wake lock is held to insure that CPU is running while the screen is off.

The DFS scheme on Android is implemented with 'ondemand' policy. In our project we apply power optimisation on cpu by controlling the ondemand policy in runtime. Details of the optimisation are discussed in Section 5.3.3.

OnDemand Governor

OnDemand is a commonly used policy on Linux environments. The high-level algorithm of ondemand policy can be summarised as follows. The algorithm dynamically controls CPU frequency in response to CPU utilisation [30]:

```
procedure Ondemand-DFS(cpu util)
 if cpu util ! UP_THRESHOLD then
   Set-Frequency(highest frequency)
 else if cpu util " DOWN_THRESHOLD then
   requested freq <- frequency that maintains a
utilization of at least UP_THRESHOLD10%
   Set-Frequency(requested freq)
 return

 procedure Set-Frequency(freq)
 if powersave_bias = 0.0 then
  Set CPU frequency to freq
 else
  Alter CPU frequency dynamically to maintain an effective frequency of:
  freq  ((1000 - powersave_bias) * 0.001)
 return
```

If CPU utilisation rises above the threshold value set in the up_threshold parameter, then ondemand governor increases the CPU frequency to scaling_max_freq. When CPU utilisation falls below this threshold, the governor decreases the frequency in steps; it sets the CPU to run at the next lowest frequency. The lowest frequency that the CPU can go is bounded by scaling_min_freq parameter. After each sampling_rate milliseconds, the current CPU utilisation is re-examined and the same algorithm is applied to dynamically adjust the CPU frequency to current process load.

During periods of low utilisation, ondemand policy aims to save power by reducing the clock rate. As we explained earlier, this strategy in some cases is not effective and might even be counterproductive. However the ondemand governor has a tuning control called powersave_bias.

The purpose of powersave_bias is to modify default behaviour of the ondemand policy. In order to save more power we can reduce the target clock rate by a specified percentage. powersave_bias property has values between 0 and 1000 that is used as % rate, to decrease the current CPU frequency. By default, (powersave_bias = 0), the ondemand governor selects the minimum processor frequency that can still complete a workload with minimal idle time. This should result in the highest performance to power efficiency ratio. In some cases, which we will discuss in Section 5.3.3, we prefer a greater emphasis on power efficiency than performance. In this case, we can control the target CPU frequency (upper bound), by setting the powersave_bias parameter to a value between 1 and 1000, thus we reduce the target frequency by one-thousandth of that value.

For example, setting powersave_bias to 100 would result in 1/10 reduction in target frequency. In this case, if the governor chooses a target frequency of 384MHz (with powersave_bias = 100), then ondemand will request 345Mhz 10% reduction. If 345Mhz is an exact match with an available hardware frequency (the scaling_available_freq parameter), then processor is set to this frequency. If 345Mhz GHz is not available, then processor alters between the closest available upper and lower frequencies for an average frequency of 345Mhz GHz.

To set the powersave_bias value, power controller writes values into a system file:

/sys/devices/system/cpu/cpu0/cpufreq/ondemand

5.2.2 LCD Screen

The screen consumes a great amount of energy in its maximum state, but unfortunately has only few power saving features to manage it. Since power consumption of the screen is proportional to the luminance it delivers [5], energy can be saved by reducing screen‘s brightness level or by turning it off. Generally, the only strategy to save power on screen, is to reduce the brightness or switch it off. Android, users can configure an inactivity timeout setting. Which will make system automatically switch off screen after defined period. Also user can adjust the brightness of the screen manually.

The above strategy is automated in PowerRunner as part of the screen's power management strategy. We have implemented screen hardware controller, which is capable to adjust the brightness level, and switch on/off states at the low level of the system.

To change the state of the screen, controller modifies a system file, with a value of desired brightness $(0 - 255)$:

/sys/class/leds/lcd-backlight/brightness

The controller also has ability to force the screen to switch off. and we are using Android‘s PowerManagerService to execute it.

Code sample:

PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
pm.goToSleep (SystemClock.uptimeMillis())

We cover the screen power optimisation techniques in Section 5.3.2

5.2.3 Wireless Network Interface

Wireless network interface has five operating modes; in order of decreasing power consumption, these are [5]:

- Transmit mode - The device is transmitting data.
- Receive mode - The device is receiving data.
- Idle mode - The device is doing neither, but the transceiver is still powered and ready to receive or transmit.
- Sleep mode - In sleep mode, the transceiver circuitry is powered down, except sometimes for a small amount of circuitry listening for incoming transmissions.
- Off - While device is off, no power is consumed.

Switching to idle mode is not very practical, because the power consumption in idle mode is not significantly less than that in receive mode. Typically there is a delay cost associated with transition between idle and sleep mode [31].

Some wireless devices provide the ability to dynamically modify their transmission power. Reducing transmission power decreases the power consumption of the transmit mode. (implementation of this optimisation is outside the scope of this project)

In the frame of hardware control for wifi interface, we have implemented a controller for transitions between idle and sleep modes. To achieve this, we used a provided API, in Android's SDK.

****

WifiManager wm = (WifiManager)Context.getSystemService(Context.WIFI_SERVICE);
wm.setWifiEnabled(boolean);

****

5.2.4 Telephony Interface

Telephony Interface has two functions: one is to provide telephony service such place/retrieve calls, and other is data services on top mobile network (GSM/EDGE/UMTS). Data service has similar properties to wifi interface.

We have implemented a controller for telephony data services. There is no public API available to control the Data connection state, so to get control we had to 'hack' the system. The modem processor switches off the functional unit of data services, when the connection string (APN) is invalid. Thus to disable data services, we modify the APN connection string and make it invalid, by simply adding a prefix. To enable it back on, we remove the prefix, and modem automatically re-establish the connection.

Android makes some of the system settings accessible via content providers (part of Android SDK) [23], we used the following to control the data services:

```
// from frameworks/base/core/java/android/provider/Telephony.java
private static final Uri CONTENT_URI = Uri.parse("content://telephony/carriers");

// from packages/providers/TelephonyProvider/TelephonyProvider.java
private static final Uri PREFERRED_APN_URI = Uri.parse("content://telephony/carriers/preferap
```

We discuss the power management technique for Data services in Section 5.3.

5.3 Power Policy Manager

As we introduced in Section 3.3, the function of policy manager is to apply effectively different power management techniques to save energy. The real time decisions about techniques and devices managed, based on the state of the system. In our project we power manage the mobile device, by observing the user's activity. In Section 5.2, we mentioned four most power hungry components, which we targeted in our project:

- CPU
- Screen
- Wifi
- Modem

In this project we adopted two types of techniques: Dynamic Power Management and Change-Blindness Optimisations.

5.3.1 User Activity and Application

In Section 3.3.1 we introduced concept of user activity. We explained that each activity can be categorised by a set of requirements and time it lasts. An application is classified as a type of Activity. For example an eBook reader app ''ALDIKO'' we can classify as ''Books'' activity. Books activity has the following properties:

- Low CPU requirement (during activity, almost no computation power needed, user is occupied with reading).
- Network is not required (eBook files are stored on disk, and when reading network is idle).
- High screen brightness (People typically prefer brighter screens when they read).

The automatic classification of applications were not implemented in this project due to lack of time. However by using a machine learning technique such as 'decision trees', a set of trees can be trained for a set of activities. And thereby classifying applications would be a trivial task.

During a runtime of Aldiko application, we can scale the CPU clock down, switch off the network, and adjust the screen with change blindness technique 5.3.2.

Some applications might have unique behaviour pattern. By learning the application pattern, we predict the timeout and idle thresholds for switching off network devices. For

example consider a mail client application, which is used by two users one is businessperson and other is average user. User1 receives emails in high frequency, while User2 receive 2-3 emails per day. By analysing the packet rate of network interface, we estimate the amount of data being delivered after idle period and adjust the timeout and off periods accordingly.

Learning Application Context

Another type of learning we do is user's feedback. During the operation PowerRunner has a status menu where user can adjust screen brightness, toggle on/off network and wifi interfaces and request to speed up cpu clock. We store those adjustments and running applications in a knowledge base and next time the application runs we take user's preferences into account, when setting the PM policy.

5.3.2 Screen Brightness

In [32], Shye et al., successfully used human psychology study of change-blindness to save energy. According to [32], a research of human psychology and perception revealed a disability of human brain to detect large changes in their surrounding environment. Shye et al. suggested a method of slowly dimming the screen to a certain level (about 70% or original value). The experiment showed an economy of 10.6% in power consumption, using this method. Change in the screen brightness likely to be unnoticeable due to the change blindness in human. And a result is economy of energy. We adopted this method in our project as follows. Policy Manager initially stores the present screen brightness level. And listen to screen change events. As long as screen is on , the Policy Manager will drop 5 units (out of 255) of brightness every 3 seconds, until it reaches 60% of the original value. When screen goes off, we reset the setting to the original value. So, when user switch screen back on, it has the preferred brightness.

We learn the blindness level by the user feedback. We keep dimming the screen up to 60% of the original setting. User has an option to adjust the screen brightness from the status menu of PowerRunner. We record the state of the screen, when user requests to adjust it. We learn the brightness level at which user notices the change (and press the button). At this stage, we restore the original brightens level, and raise the blindness level slightly higher (3%). The rationale behind this is to adjust the mobile device to the blindness level of a user and benefit from energy saving of the screen.

5.3.3 CPU Dynamic Frequency Scaling

As we explained in Section 5.2.1, Android OS uses OnDemand DFS policy to manage the power of the CPU. We also highlighted the ineffectiveness of scaling the clock of the CPU. However by knowing the computation requirements of an application, we optimise the Ondemand policy for more effective power management. Consider again an ebook reader application. We know device doesn't require high CPU frequency because user is reading an ebook, which doesn't require computational power. Given the fact the device will remain on for the whole time user reading the ebook, CPU will be operational as well. Thus once we identify this application running, we can scale the cpu clock down. Another example: consider a phone application (the system app, which place and retrieve calls). We know that during phone call activity user mainly occupied by the phone call. This application works perfectly in low CPU frequency, thus once policy manager receives notification *CALL_STATE_OFFHOOK*, it considers scaling CPU down. The only condition which might stop from slowing the clock, is the number of applications running. In multi-threaded environment like Android, user can run several applications. Android Os manage the resources automatically, and might even kill unused processes in low memory situations. We analyse the activity of each of the running applications. And if we identify more than one active app, we switch the ondemand to default mode (utilisation based).

On-Demand Policy

We tune the ondemand policy by changing parameters which are held in system files at */sys/devices/system/cpu/cpu0/cpufreq/ondemand* directory.

To scale the clock down, we are setting *sampling_rate_max* to be as *sampling_rate_min*. We also learning by the user‘s feedback the lowest upper bound of the CPU at which user is satisfied with performance. During a normal operation, from the moment screen switched on, every five seconds policy manager increases the *powersave_bias* parameter by 20 units (decrease current clock rate by 2%). This is until a maximum of 400 units reached (60% of maximum frequency requested by ondemand). If user feedback request for higher performance (presses ‘’faster‘’ button, in the status menu), we reset the maximum frequency to default settings, adjust the decrease level (400 units), by 50 units and start the process again, until the maximum level is found, and user remains satisfied with performance. This gradual CPU scale down process, we run every time user switches on a screen (come back from idle). When device goes idle, we reset settings to default values. This method exploits the change-blindness effect, and also common behaviour of mobile device users.

5.3.4 Network

As we manage the state of network interface, we begin with default policy of *inactiviy_treshhold=60 sec* and *off_period=5 min* (We switch off network after 60 seconds of inactivity for 5 minutes). We learn applications network requirements by analysing the network traffic, after network interface is switched on. We then adjust the timeout and off period to minimise the network activity after the off intervals. We also learn the time intervals from user's feedback. If user requests to switch on network, which its off, we readjust the times. This happens until user is satisfied with the level of service (No delay in network activity).

APN

In Section 5.2.4, we presented a method for switching the Data Services off.

5.3.5 Wireless LAN

Wireless network optimisation technique is similar to data services. Wireless though is cheaper (in terms of power consumption), therefore it is preferred. When we identify wireless network, we turn the network data services off (by default modem is in idle state).

Location based wifi connection

Default behaviour of wireless device is to search for wireless networks, which is costly and wasteful. In our project we implemented a policy for wireless network interface based on geo position of the mobile device. It can be summarised as follows: Every time device is connected to wifi network, we store the current geo position and the network id in sqlite db (part of Android platform). Policy manager listens to location events and when the device outside the range of any known network, PM switches off wifi interface.

5.4 Summary

In PowerRunner we have implemented the three fundamental modules of the power management framework. Those are: The Controller, Observer and the Policy Manager. In the scope of Controller module we have implemented control functions for CPU , Screen, Wifi and Modem components. As part of the Observer module, we implemented a set of monitors for majority of the hardware devices on HTC G1. In Policy Manager we employed DPM

and Change-Blindness techniques to effectively save energy. Policy Manager's function is to analyse and learn user feedback and applications usage pattern, for effective utilisation of dynamic power management techniques. PM also uses geo-awareness to control the state of wifi interface.

Chapter 6

Evaluation

As stated in our introduction this project had two major goals:

- The first is to design intelligent modular power management architecture suitable for any mobile platform. It will be able to learn how each particular mobile device is used. Analyse gathered data and define the most efficient power management strategy for that particular mobile device. As a result, a consumer will receive a customised power management solution best suitable for his/her needs.
- The second is to create a generic testing framework, which will enable developers to test their applications for power consumption. The framework will analyse what internal devices are used by the application. Then, by using a linear regression based model for estimation of power consumption, it will transfer information into power units. It will help developers to height leaks and possible savings of power.

The aim of the evaluation phase is to evaluate if the implementation achieves the goals set before the start of the implementation and if our architecture can be used in practice. The ultimate goal of a power management framework is to reduce the power consumption of a system. Our power management framework is fundamentally based on prediction of resources required by a specific application. Thus to prove the efficiency of our framework we defined a suite of tests, which utilises a set of applications. Each application has unique hardware requirements. During the test session, we expected to see a significant reduction in power consumption.

6.1 Test Setup

We developed the PowerRunner with a simple User Interface. PowerRunner run in two modes: Profiler and Optimiser. Figure 6.1 illustrates the main menu of the application.

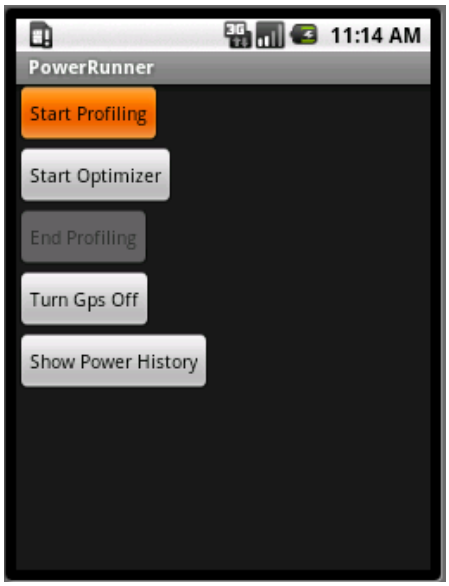


Figure 6.1: PowerRunner main menu

To run the PowerRunner as power manager, user need to press the *Start Optimizer* button. Power Runner then goes into background and user is taken to the home menu, where he can start any any activity. During the run PowerRunner identify running application and automatically manage the states of the devices as we discussed previously. It also records user power consumption. To evaluate the energy saving, we ran a series of activities, with and without the optimiser. We ran several cycles of the test sequence with different screen brightness settings. The sequence of activities was as following:

1. CPU Benchmark utility, calculates π : test normally runs about a minute
2. GPU benchmark tool, provided by Qualcomm. Tool is testing the graphic functional unit of the ARM11 processor. Test runs approximately 5 minutes, and renders a 3D animation. We used this test to emulate gaming activity.
3. Network Test. We used Network Benchmark utility, which tests the bandwidth of the connection. It uploads and downloads 5Mb file test runs few minutes
4. Music listening activity. We used a default music application, which plays an mp3

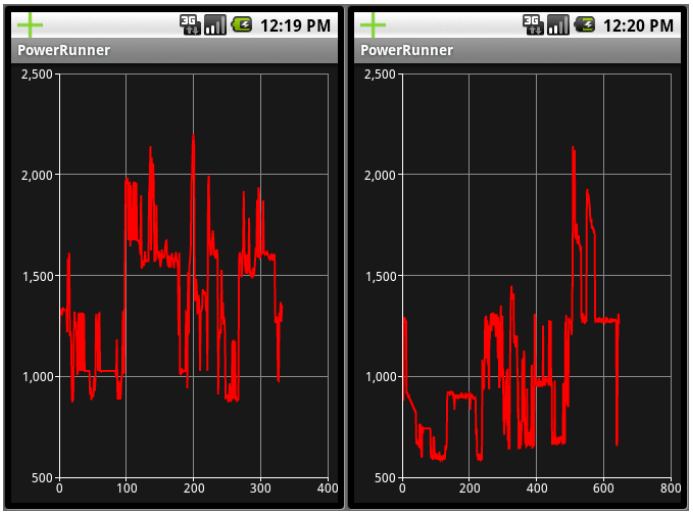
file of the SD card. Test runs for 5 minutes

5. SD Card stress test. We used an stress testing application, which run a series of test to benchmark read/write performance of the SD card. Test runs for 10 minutes.
6. Phone Call Activity. We emulated a typical phone call activity, by calling to the tested device. We run several test to emulate when the phone is ringing and when the call is connected. During the interval when the call was in progress we switch the speakerphone on and off periodically to capture the various energy consumptions.

6.2 Results

We used our power estimation model to validate the effectiveness of the power management framework. The PowerRunner's testing framework was used to illustrate the results of power management.

Our results indicates that the power management framework successfully saved 25% energy. In figure 6.2 shown the comparison between a normal operation and operation with power management turned on. Under unmanaged operation a series of tests consumed on average 1391 mW. Running the same test with power optimisation turned on, the system consumed an average 1043mW.



(a) Normal power consumption (Power mean: 1391mW)

(b) Optimised power consumption (Power mean: 1043mW)

Figure 6.2: Power consumption comparison

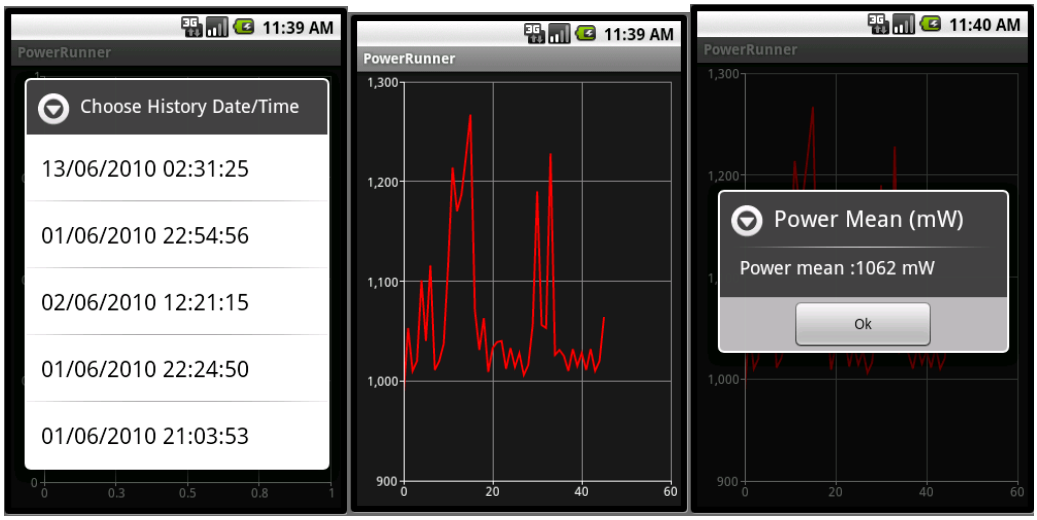
Overall, our results show that :

1. Our power estimation model can accurately predict the power consumption of the total system
2. The power management framework, successfully minimised the total energy consumption of the system.

6.3 PowerRunner as testing tool

Additionally to the power management, developers can use PowerRunner as a testing tool. We now introduce an intuitive workflow for power consumption testing of an application. On main menu shown in figure 6.1, developer has an option to run profiler. Once the 'Start Profiling' button is pressed, PowerRunner start profiling the system and the user is taken to the home menu of the phone where he can start testing his application. During the runtime of the tested application, PowerRunner collects measurements and estimates power consumption of the hardware components. Also during runtime, the estimated power consumption is presented on the status bar. Once user finishes the testing of his application, he can use an icon in the status bar to go back to PowerRunner, where he can choose to stop profiler. The whole session is recorded in memory. The session id flushed to disk when profiler is stopped. User now can choose the 'Show Power History' button from the main menu. The history activity within the PowerRunner present the user with a list of historic recordings. From that list user choose the latest to view a chart of power consumption for his application. The chart has an option to show the mean of the power consumption in milliwatts.

Figure 6.3 illustrates a utility within PowerRunner to review the history and the mean of the power consumption.



(a) User History (b) Power Consumption Chart (c) Total mean

Figure 6.3: Power consumption history in PowerRunner

Chapter 7

Conclusion

In this section, we do review the goals we have set yourself in the beginning of the project and appreciate how we reached them, and to what extent. Based on what we reach, we suggest some areas where further work could be done to expand our work and solve the problem of power management.

7.1 Review of Goals and Achievements

Looking back at Chapter 1, the goals of our project were to:

1. Develop a power management architecture suitable for any mobile platform.
2. Create a generic framework for testing of applications for power consumption.

Now, at the end of our project, we have reached our goal of almost fully. We have shown that, by learning the resource utilisation of user's activities we have been able to predict and supply the minimum required resources and thereby successfully save energy on mobile device. In this project, we proposed a unique power management framework which leverage the user's behaviour to save energy. Additionally we created an accurate power estimation model, which allow to estimate power consumption of individual components and applications in real time. Validation tests indicates that the model is highly accurate. The estimation model can be used for estimating current consumption and forecasting the future resource availability.

We implemented the proposed architecture on Android OS, proving that theoretical design could be implemented practically. We also evaluated the software to verify that the framework does save power.The test results were positive; Therefore, we believe that our strategy

of power management can be applied in practice and would reduce the energy used throughout the system.

7.2 Limitation

Due to lack of time, we haven't implemented the automatic classification of applications.

7.3 Future Work

The next step for this project is to implement the classification mechanism. Integrate Ponder2 policy evaluation framework. In terms of further validation of the framework, the application is to be published and real usage statistics to be collected.

Bibliography

[1] J. R. Lorch and A. J. Smith, "Software strategies for portable computer energy management," Berkeley, CA, USA, Tech. Rep., 1997.

[2] Google, "Coding for life – battery life, that is," Website, http://dl.google.com/io/2009/pres/W_0300_CodingforLife-BatteryLifeThatIs.pdf , Accessed: 15 January 2010.

[3] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," pp. 231–248, 2002.

[4] S. Gary, P. Ippolito, G. Gerosa, C. Dietz, J. Eno, and H. Sanchez, "Powerpc 603, a microprocessor for portable computers," *IEEE Des. Test*, vol. 11, no. 4, pp. 14–23, 1994.

[5] E. Harris, S. Depp, W. Pence, S. Kirkpatrick, M. Sri-Jayantha, and R. Troutman, "Technology directions for portable computers," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 636 –658, apr 1995.

[6] M. Stemm, M. Stemm, R. H. Katz, and Y. H. Katz, "Measuring and reducing energy consumption of network interfaces in hand-held devices," 1997.

[7] L. Benini and G. d. Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[8] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1995, p. 374.

[9] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 1998, pp. 197–202.

[10] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1999, pp. 134–139.

[11] C. Hwang and A. Wu, "A predictive system shutdown method for energy saving of event-driven computation," *ACM Transactions on Design Automation of ...*, Jan 2000.

[12] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 4, no. 1, pp. 42–55, 1996.

[13] A. Bogliolo, G. Paleologo, and G. D. Micheli, "Policy optimization for dynamic power management," *IEEE Transactions on ...*, Jan 1999.

[14] T. Simunic, L. Benini, and G. D. Micheli, "Energy-efficient design of battery-powered embedded systems," *Proceedings of the 1999 ...*, Jan 1999.

[15] E. Chung, L. Benini, and G. D. Micheli, "Dynamic power management using adaptive learning tree," *Proceedings of the 1999 IEEE/ ...*, Jan 1999.

[16] ACPI, "Advanced configuration and power interface specification. the acpi specification - revision 4.0."

[17] LifSoft, "Acpi faq," Website, http://www.lifsoft.com/power/faq.htm , Accessed: 15 January 2010.

[18] Kernel.org, "Linux power management support."

[19] N. Kappiah, V. Freeh, and D. Lowenthal, "Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs," *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Nov 2005.

[20] L. Benini, A. Bogliolo, S. Cavallucci, and B. Riccó, "Monitoring system activity for os-directed dynamic power management," *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, Aug 1998.

[21] Wikipedia, "The free encyclopedia," Website, http://en.wikipedia.org , Accessed: 15 May 2010.

[22] W. Bircher, M. Valluri, J. Law, and L. John, "Runtime identification of microprocessor energy saving opportunities," *Proceedings of the 2005 ...*, Jan 2005.

[23] E. Burnette, *Hello, Android!* The Pragmatic Programmers, 2009.

[24] B. Blog, "Into android," Online, 2010 2010.

[25] (2010, Jan). [Online]. Available: http://www.arm.com/products/processors/classic/arm11/index.php

[26] *Sata Specification*.

[27] Google, "Android sdk," Website, http://developer.android.com/index.html , Accessed: 15 January 2010.

[28] LinuxHowTo, "procstat explained," Website, http://www.linuxhowtos.org/System/procstat.htm , Accessed: 15 May 2010.

[29] J. Lorch and J. Lorch, "A complete picture of the energy consumption of a portable computer," 1995.

[30] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *In Proc. of the Linux Symposium*, vol. 2, 2006.

[31] G. Anastasi, M. Conti, E. Gregori, and A. Passarella, "802.11 power-saving mode for mobile computing in wi-fi hotspots: limitations, enhancements and open issues," *Wireless Networks*, vol. 14, no. 6, Dec 2008.

[32] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 168–178.