1-1-2014

Energy Efficiency Analysis And Optimization For Mobile Platforms

Grace Metri
Wayne State University,

ENERGY EFFICIENCY ANALYSIS AND OPTIMIZATION FOR MOBILE PLATFORMS

by

GRACE CAMILLE METRI

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2014

MAJOR: COMPUTER SCIENCE

Approved by:

____

Advisor Date

____

____

____

____

DEDICATION

To my beloved daughter, Katherine

and

To my parents, Camil and Rose

For allowing me to dream with no limits and teaching me to be persistent and to always aim higher; And most importantly, for their unwavering love and support.

ACKNOWLEDGMENTS

I am humbled by the large number of people who have directly or indirectly helped me reach my goal of completing my Ph.D. dissertation. First of all, I would like to thank my academic advisor Dr. Monica Brockmeyer who started mentoring me when I was still an undergraduate student. Meeting her was a life-changing event. She is the one who encouraged me to join the graduate program and provided support throughout my academic journey. Her dedication, guidance, faith in me, and understanding were crucial for attaining my goals, and for that, I am forever grateful.

I would also like to thank my co-advisor, Dr. Weisong Shi. I also knew him as an undergraduate student and attended several courses taught by him. I was always impressed by his level of knowledge and his style of fostering a researcher mentality, all of which are great qualities of an advisor. His support, advice, and flexibility were very important to my progress in the graduate program. My gratitude for all the time an energy he invested in me is beyond words.

I would also like to thank Dr. Daniel Grosu and Dr. Nathan Fisher, from the Computer Science Department at Wayne State, for serving as committee members during my prospectus and dissertation defense and providing me with excellent detailed feedback on my work and dissertation. I would also like to thank my external committee member from Intel, Dr. Kyung Hee Kim, for her great support, encouragement, flexibility, and feedback on my dissertation projects.

I am happy that I was part of the Department of Computer Science at Wayne State for so many years. I am thankful for all the staff and faculty members. In particular, I am very grateful for the help and support that I received from Dr. Seymour Wolfson and Dr. Farshad Fotouhi when they were both the department chairs. They always had their door open and provided help and support when needed. I am also proud to be part of the MIST group and work among some of the smartest and most driven Ph.D. and Masters students.

In addition, I have been very fortunate to work for Intel for the past couple of years. I was exposed to some of the best cutting-edge technology and tools which made this dissertation possible. I am particularly thankful for my mentor, Abhishek Agrawal. The impact that Abhishek had on my research and career development is priceless. Words simply cannot express my gratitude for all the time he spent mentoring me and supporting me to achieve my research goals. I would also like to thank my colleagues and my current manager, Karla Callaghan. Her support, encouragement, and flexibility throughout the past few months were the driving force behind completing the last milestone of my dissertation.

I also want to thank my friends for their support and love. In particular, I am very thankful for Sonia Haiduc. I don't know what I would have done without your continuous support and occasional surprises. My dear Lily Lee, thank you for encouraging me to follow my dreams. Paul Janiczek, thank you for helping me discover Detroit and all its hidden authentic restaurants and, of course, helping me with all the logistics in preparation for my defense. Talia Selitsky, Katie Kunz, Claudia Valdes, Melissa Guinn, Michele Donato, Laura Moreno, Radu Vanciu, and Cristina Mitrea thank you for being there whenever I needed a break. You are all true friends.

Finally, I am blessed by my great family. Everyone had an impact, one way or another, on my life. First, and foremost, my grandmother *teta* Daad Nohra. All I can say is that I wish someday, I will be half the great woman you are. My dear parents, Camil and Rose, I thank you for all your sacrifices, help, support, encouragement, and love. My siblings, Joanne, Ralph, and Jeffery, thank you for being my best friends despite all the distance. My aunt, Liliane Nohra, thank you for inspiring me when I was still a teenager to be just like you when I grow up. Throughout my journey, you provided unlimited support and encouragement. You read all my research papers and provided feedback even though the subject of my research is completely foreign to you. So, thank you for all your efforts. My aunt, Maggie Nohra, thanks for all your love and support since I moved to the states. My cousins, Dr. Mikhail Mitri and Fr.

Antonio Feghali thank you for listening to my occasional complaints and extending your full support and love. Elie Samaha, for helping me raise our beautiful daughter, Katherine. Finally, I am mostly thankful for you, Katherine. We did it, dear!! I think we equally sacrificed since the beginning of this journey. You gladly accompanied me to the lab or meetings on many occasions instead of going to play dates and you made sure to help me with numerous papers I worked on by giving them life through your colorful drawings. I love you!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The introduction of mobile devices changed the landscape of computing. Gradually, these mobile devices are replacing traditional personal computers (PCs) to become the device of choice for entertainment, connectivity, and productivity. Everyday users use their mobile devices daily to play games, pay a bill, or make a phone call. Businesses are using them to receive payments or to enable consumers to place orders in cafes and restaurants.

There are currently at least 45.5 million people in the United States who own a mobile device, and that number is expected to increase to 1.5 billion by 2015 [80]. Users of mobile devices expect and mandate that their mobile devices have maximized performance while consuming minimal possible power. Users don't care if the hardware is optimized for maximum energy efficiency or if the software is maximized for energy efficiency. When they evaluate their devices, they look at the overall battery life of their devices. However, due to the battery size constraints, the amount of energy stored in these devices is limited and is only growing by 5% annually [73]. As a result, we need to analyze the energy efficiency of these mobile devices and use the lessons learned in order to optimize the energy consumption and thus increase their energy efficiency.

Hardware manufacturers came a long way into reducing the power consumption of their platforms, but optimization of energy efficiency will not be attained unless applications (apps) that are running on these platforms are optimized in terms of energy efficiency as well. As a matter of fact, any mobile platform consists of three layers: the application layer, the management layer, and the hardware layer, as shown in Figure 1.1.

- **The Application Layer:** This layer consists of the applications running on a platform. These applications can impact the power consumption of the device based on their resource utilization.

- **The Hardware Layer:** This layer consists of a collection of physical resources included in the platform such as display, Wi-Fi radio, sensors, and cameras. The number of physical resources and their power state affect the overall power consumption of the device.

- **The Management Layer:** This layer contains the algorithms and policies that perform the resource allocation required by the application layer to the hardware layer. It also performs power management of the physical layer by changing the power states of each physical resource.

Because the mobile device is a single unit, in order to achieve optimized energy efficiency, all three layers must be optimized in terms of energy efficiency, because one misbehaving layer can affect the entire unit. They need to work in sync as a single unit to achieve this goal. This means that the application should be created using energy-aware algorithms in order to minimize the use of physical resources. The hardware components should consume the least amount of possible power when in use and be capable of switching their unused components to low-power states (or maybe shutdown mode) when not in use, in addition to reducing the tail power of components such as the tail power of Wi-Fi radio. The power management layer should map the application tasks to the appropriate physical resource (e.g., allocation to the appropriate intellectual property (IP) in terms of the power management layer of SoCs). In addition, based on the workload of applications, it should change the power states of physical components.

1.1 Research Goals

Our research goal is to analyze the energy efficiency of mobile devices. Then, use the lessons learned from our analysis in order to increase the

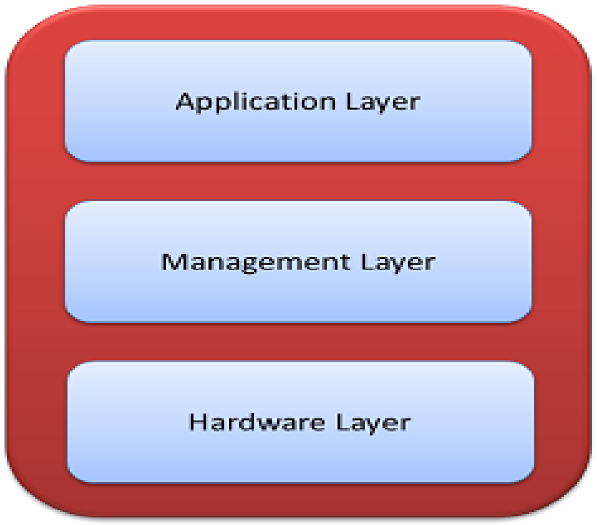


Figure 1.1: Mobile devices architecture.

energy efficiency of mobile devices.

Analysis of energy consumption of mobile devices is a very complicated process but it is key to optimizing their energy efficiency. Without proper understanding of how power is dissipated in a platform, we cannot increase its energy efficiency. That means we need to have appropriate profiling tools to determine the power dissipation of different physical components. In addition, we need to be capable of mapping the power consumption behavior of physical resources to the execution behavior of applications. Moreover, having extensive amount of power metrics collected without full understanding of the cause and effect of the differences in power consumption behavior and how one metric's power consumption is affecting the power consumption behavior of a different metric is ineffective to determine the causes of energy inefficiencies. To this extent, we approached solving our research goals through the following key directions:

1. **Developing a tool in order to determine the behavior of power consumption of the CPU with negligible overhead.** In order to be able to understand the power dissipation of physical components of mobile devices, we need to develop tools which are capable of not simply providing total power

consumption of a component, but the power consumption behavior. Since energy efficiency of a platform cannot be increased without understanding how the application layer affects the physical layer, then by extracting the power consumption behavior of the hardware and correlating it with the execution of applications, we are enabled to determine effectiveness of the power management layer of the platform and the energy inefficiencies of apps.

2. **Providing techniques to increase the energy efficiency of apps.** Since the energy efficiency of mobile devices is highly dependent on the energy efficiency of the running apps, it is critical for developers to be aware of techniques to increase the energy efficiency of their apps. Our goal is to provide energy efficiency development rules which need to be followed by app developers. In addition, our goal is to find the gaps of why there are still many popular apps that are energy inefficient. Finally, we want to demonstrate proper means for developers to profile their apps and determine the causes of its inefficiency.

3. **Determining energy consumption issues when smartphone devices are in idle state and proposing optimization techniques.** Smartphones are usually idle for the majority of the battery life duration. However, they need to remain connected to a network at all times in order to receive notifications and updates. Since smartphones allow background applications to run during idle time, our goal is to determine the impact of background applications, based on their category, on the overall energy consumption of a smartphone in addition to determining the impact of network connection type (3G versus Wi-Fi) on the overall energy efficiency. By understanding the impact of background applications and network connection type on the energy efficiency of a smartphone, we can provide recommendations in order to increase the energy efficiency of smartphones at idle time.

4. **Providing a tool to extend battery life on demand.** Our goal is to enable

users to extend battery life on demand for a specific duration until a particular task is performed.

5. **Analyzing power consumption behavior of Systems-on-Chips (SoCs) and providing techniques in order to increase their energy efficiency.** Current mobile devices are using SoCs which contain in addition to the cores, specialized custom engines. One of the advantages of these custom engines is enabling the CPU to offload part of the execution to these specialized engines. Our goal is to examine the impact of offloading tasks to the engines on the energy efficiency of mobile devices. In addition, our goal is to highlight techniques to optimize the SoCs from an energy efficiency perspective.

1.2 Our Approach and Contribution

In this section, we summarize our approach to accomplish our research goals and in addition to the summary of our contribution.

1. **Developing a tool in order to profile the power consumption of the CPU.** We developed *SoftPowerMon*, a power-profiling tool for Android mobile devices. We created two flavors of the tool: one which runs on a host system and the other one runs as a native app on the Android device. The tool can be used to determine "why" a specific amount of power was consumed as opposed to "how much." It consumes negligible overhead and does not require flashing of the kernel. One of the key advantages of this tool is that it can collect data on any type of processor. By using *SoftPowerMon*, platform manufacturers can strictly power profile the processor without taking into consideration any other device component of the platform. Thus, they can determine the energy efficiency of one processor compared to another. In addition, app developers can use it in order to observe the impact of their apps on the energy efficiency of the processor. In order to highlight the features and capabilities of *SoftPowerMon*,

we provide case studies using smartphones and tablets. Using the case studies, we were able to highlight the benefits of the tool. We present this tool in Chapter 3. A paper describing this tool was published in the Proceedings of the International Conference on Energy Aware Computing 2012 [71].

2. **Providing techniques to increase the energy efficiency of apps.** We developed a tool called *EnergyMeter* which can collect the platform, package, core, and GPU energy consumption on Windows platforms. Then we characterized the mobile apps into eight categories: browsers, video streaming, music streaming, maps, video chatting, social networking, and email services. Then, for each mobile app category, we profiled the energy efficiency of app collection on the three most popular operating systems: Windows, iOS, and Android, in terms of: 1) same app categories on a single OS compared to other apps of the same category and on the same OS. 2) Ranking the energy efficiency of the same app on different OS. 3) Comparing the energy efficiency of the same application if accessed using a native app or using a browser. Based on the case studies, we derived a list of observations, causes, and implications. Finally, we provided a list of the top 10 energy efficiency app development rules which are recommended for app developers. We present this project in Chapter 4. A paper containing a subset of our case studies has been submitted to the 5th International Green Computing Conference (IGCC'14) and is currently under review. The full report including raw data will be available online.

3. **Determining what is eating up battery life on smartphone devices when idle.** In order to achieve this goal, we provide detailed case studies on the two popular smartphone devices: an iPhone (iOS) and a Samsung S3 (Android). We observed the impact of background applications and network connection type (Wi-Fi versus 3G) on the overall energy consumption of the device. These case studies are particularly important because focus in literature

has been on active workload on mobile devices. However, smartphones remain in extended duration in idle state. Therefore, it is important to determine energy inefficiencies due to background apps. In addition, through our case studies, we were able to derive a list of optimization techniques which can extend the battery life of smartphone devices. Finally, we showed that even though some concepts are widely known to increase the energy efficiency of smartphones, however, the techniques were not adopted by the two most popular platforms. Thus, there is still room for improvement. We present this project in Chapter 5. A paper describing this project was published in the Proceedings of the International Conference on Energy Aware Computing 2012 [72].

4. **Providing a tool to extend battery life on demand.** We developed *BatteryExtender*, an adaptive user-guided tool for power management of mobile devices. The tool enables users to reconfigure the device's resources based on the user's workload requirement, similar to the principle of creating virtual machines in cloud environments. The tool can predict the battery life savings based on the new configuration, in addition to predicting the impact of running applications on the battery life. Through our experimental analysis, BatteryExtender enabled users to decrease the energy consumption of mobile devices between 10.03% and 20.21%, and in rare cases by up to 72.83%. The accuracy rate ranged between 92.37% and 99.72%. We present this tool in Chapter 6. A paper describing this tool was published at the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2014).

5. **Analyzing power consumption behavior of Systems-on-Chips (SoCs) and providing techniques in order to increase their energy efficiency.** In order to achieve this goal, we highlight the importance of offloading task execution from the CPU to dedicated IP blocks by showing how it can be effec-

tive in energy optimization of SoC devices. We provide supporting data for our claims by including thermal images of an SoC while offloading was enabled and while it was disabled. We also make a strong case for new power-profiling tools that take a holistic view of the systems, including peripherals and accelerators that are beyond the CPU. We provide two case studies, one using GPU/CPU for video decoding and one using DSP/CPU for audio decoding, to show that today's SoC devices require very fine and sophisticated power-profiling tools to account for the SoC's exercised offloading mechanism of functionality to different IP blocks. Finally, we show that current software-based power-profiling techniques for SoCs can provide an error rate close to 12%. Thus, they cannot be used for increasing the energy efficiency of workload which offload form CPU to the dedicated IP blocks. We present this project in Chapter 7. Two papers were published for this project. The first one was published at the 4th Annual International Conference on Energy Aware Computing Systems and Applications 2013 [87] and the second one was published at *Computer* [70].

1.3 Organization

The remainder of this dissertation is organized as follows: in Chapter 2, we give some background information related to our work which includes a clear distinction between energy and power, description of power metrics that affect the processor's power consumption in addition to energy overhead analysis of the three layers of a mobile device: the hardware, management, and application layers. We also present a discussion of the related work in the existing literature related to energy efficiency analysis and optimization of mobile devices. In Chapter 3, we describe *SoftPowerMon*, a profiling tool for Android mobile devices, along with case studies that highlight the tool's usefulness. In Chapter 4, we focus on energy efficiency comparison of mobile platforms through a quantitative approach. We describe *EnergyMeter*, a profiling tool to collect energy consumption on Windows-based platforms. Then, we collect power metrics for eight common usage scenarios of mobile devices on the three most popular platforms: Windows 8 and beyond, iOS, and Android. Based on the interpretation of the results of our case studies, we derive a list of implications which can be used by app developers in order to optimize the energy efficiency of their apps. In Chapter 5, we focus on energy profiling of background applications running on smartphone devices (while the device is in standby) on two different mobile platforms: iOS and Android. We also analyze the differences in energy consumption based on network connectivity type (Wi-Fi versus 3G). Finally, we use knowledge gained through our analysis in order to provide a list of implications which enable developers to improve the energy efficiency of their apps. In Chapter 6, we present *BatteryExtender*, an adaptive user-guided tool for power management of mobile devices. The tool enables users to extend battery life on demand for a specific duration by reconfiguring the device's resources based on the user's workload requirements and by profiling the energy consumption of apps and predicting the battery life savings. In Chapter 7, we describe SoC devices and concept of offloading to a specialized engine. Then, we provide optimization techniques in order to increase the energy efficiency of the SoC and show the gap in available tools to accurately profile such devices. Finally, in Chapter 8, we present our conclusion and future work.

CHAPTER 2: BACKGROUND

In this chapter, we make a clear distinction between power versus energy. In addition, we discuss the metrics affecting power consumption of processors followed by an energy overhead analysis of mobile devices. Then, we present the literature survey related to our work and conclude the chapter.

2.1 Power versus Energy

Power and energy are sometimes used interchangeably in literature. However, these two terms are far from synonymous.
Power is defined as the rate of doing work and is measured in watts (W). It is calculated as shown in equation 2.1 where Power (P) is the ratio of Work (W) over Time (T). **Electrical Power** is defined as the rate at which electrical energy is transformed to another form of energy. It is calculated as shown in equation 2.2 where power is the product of electrical current (I) and voltage drop (V) measured in watts, amperes, and volts respectively. On the other hand, **energy** is defined as the amount of power consumed over time and is measured in joules (J). It is calculated as shown in equation 2.3 where energy (E) is the product of power (P) and time (T) as shown .

$$P = \frac{W}{T} \tag{2.1}$$

$$P = I * V \tag{2.2}$$

$$E = P * T \tag{2.3}$$

A system's **Energy Efficiency** is defined as the required amount of energy needed to complete a specific workload. The energy-efficiency (EE) value is calculated as shown in equation 2.4 is the ratio of completed workload (W) over energy (E).

$$EE = \frac{W}{E} \tag{2.4}$$

A data center's **power usage efficiency** (PUE) is the ratio of total power of the facility P$_{Facility}$ over the total power of IT equipment P$_{IT_{equipment}}$, as shown in equation 2.5.

$$PUE = \frac{P_{Facility}}{P_{IT_{equipment}}} \tag{2.5}$$

2.2 Metrics Affecting Power Consumption of Processors

In order to properly power-profile a processor and isolate its power consumption from the overall host system, profiling tools should focus on metrics specific to the CPU, which determine its power consumption.

2.2.1 Processor Idle Sleep States

Modern processors attempt to reduce their power consumption by supporting different idle states known as C-states. The power savings is achieved by turning off the CPU's unused architecture blocks. The C-states supported by a CPU depend on the type of CPU. Regardless of the CPU type, the deeper the idle state, the greater the power savings.

2.2.2 Processor Performance States

Power performance states known as P-states are the processor's frequency and/or voltage. This number of frequency states depends on the processor type. The processor can change its frequency and/or voltage based on the workload as a means of saving power. The slower the processor's frequency, the less power it consumes, and vice versa.

2.3 Energy Overhead Analysis of Mobile Devices

A mobile platform architecture contains three layers: an application layer, power-management layer, and hardware layer. Each layer can impact the overall energy consumption on the basis of the following factors:

2.3.1 Application Layer

Applications running on a platform can increase the overall power consumption based on their utilization of resources. Rivoire *et al.* [84] evaluated the relationship between resource utilization and system-level power consumption on multiple platforms ranging from laptops to a server. They showed that models based on OS utilization metrics and CPU performance counters are in most cases the most accurate.

In order for developers to increase the energy efficiency of their applications, they must execute their task as quickly as possible and then enable the platform to go to idle quickly. One way to achieve this goal is by multithreading in a multicore platform where tasks can execute concurrently on each core. This will enable cores to go to idle sleep state faster than they would using consecutive execution of tasks. The second technique for developers to reduce the power consumption of their applications is by minimizing the resource utilization. This can be achieved by minimizing data movement and the efficient use of cache. In our previous work [86], we demonstrate how prefetching and caching in a DVD playback scenario can reduce disk spin, leading to a decrease in energy consumption compared to the absence of prefetching and caching.

2.3.2 Management Layer

A mobile device management layer can greatly impact the battery life by managing the power consumption of numerous hardware components. For instance, it can increase the energy efficiency of a platform by suspending the hard disk based on its utilization. It can also change the processor frequency based on its load. A lower frequency leads to lower power consumption and decreases the processor's performance, and vice versa. Moreover, the management layer can also change the processor idle sleep states, also known as C-States from active to idle. The deeper the sleep state, the lower the power consumption and the greater the transition time from idle to active and vice versa [71]. Another way to manage power consumption of the platform is by changing the platform's device power states, known as D-States, following the Advanced Configuration and Power Interface (ACPI) specifications. The device D-States enable power management of the platform to change the device's power consumption state. When a device components is in use, it is in full active state. However, when it is not in use, ideally, it is supposed to go to an idle state. There are multiple idle states, which can range from D1 to Dx. The deeper the idle state, the lower the power consumption of the device and the greater the latency to go to an active state, and vice versa. Even when a device is in idle state (i.e., not used) it still consumes various amounts of power depending on the device type. As a result, the only way to completely eliminate the power consumption of unused device components is by completely shutting off the device component.

2.3.3 Hardware Layer

The number of components in the the hardware layer depends on the platform itself. We can broadly categorize them into the following categories: processor, memory, storage, network devices, sensors, utility devices, and display. (Power consumption of the processor was already discussed in Section 2.2.)

Memory and Storage: Memory and storage power consumption depend on the number of read and write instructions. As a result, lowering the numbers of read and writes leads to lower energy consumption.

Network devices: Wireless network (Wi-Fi), Bluetooth, and Near Field Communication (NFC) are under the network devices umbrella, where Wi-Fi is proved by various literature (for example by Carroll and Heiser [32]) as the most power-consuming device in this category despite having four power states: low idle, high idle, low transmission, and high transmission. Some factors that can impact its power consumption are the network strength, upload and download data size, and its utilization frequency. This is due to the tail power, as described by Pathak *et al.* [80]. It is worth noting that the network adapter may be actively utilized by users when surfing the web, downloading material, or actively streaming videos, but it can also be triggered by background applications as well. In our previous work [72], we provided case studies on Android and iOS where we showed that when Wi-Fi was on, background applications periodically triggered data fetch, leading to an increase of battery consumption of the platform. Likewise, we should expect similar behavior on Windows platforms. More specifically, the Metro App paradigm consisting of tiles, enables its developers to create ”live tiles” that can be periodically updated [13]. A misbehaving application can frequently update the live tile, leading to an increase in energy consumption.

Sensors: Nowadays, mobile devices are built with an extensive number of sensors. For instance, Microsoft requires for all its 8.1 Ultrabooks and tablets a set of integrated physical sensors with object-oriented abstractions. The required sensors are accelerometers, gyroscopes, ambient light, compass, and GPS [54]. These five physical sensors map to a greater number of logical sensors known as fusion sensors by combining the output of multiple physical sensors. The fusion sensors are the device orientation and inclinometer. These sensors can be utilized for different purposes. For instance, the Accelelometer can be used to determine movement and speed. The compass and gyrometer improve location sensing by improving positioning through accurate direction and orientation, thus enhancing data transmitted to location-based applications. Developers can modify the update frequency of these sensors. As a result, an energy-inefficient application can keep theses sensors in active state for an extended duration by changing the frequency update interval to a very low number.

Utility devices: Utility devices have specific functionality and can be turned on/off on demand. Cameras, microphones, and speakers can fall into this category. A recent patent for Samsung Electronics Co. LT [60] transformed the usability of a camera device from a utility device, which strictly records videos or takes snapshots, to a sensing device. The patent states that the technology allows them to use the camera within a mobile device in order to acquire images, divide the images into photograph regions, and determine if the image corresponds to a command, and if so, carry out an action that changes the user interface (UI) without the need to touch the screen.

Display: The display type of recent mobile devices is either OLED or LCD. Display is one component that can significantly drain the battery. The two telemetries that can impact display power usage are the display brightness and display refresh rate. Regarding the display brightness, the power consumption of OLED displays depends on the color of the screen content, whereas the LCD display's power consumption varies based on brightness. However, according to Dong *et al.* [39], the energy contribution by OLED while updating is close to the average energy consumption of the display while the screen is constant. The second telemetry for displays is the refresh rate, which can be measured in hertz. It is basically the number of times per second by which the display hardware updates its buffer. The higher the refresh rate, the lower the flickering of images, and the greater the power consumption. On the other hand, a low refresh rate can result in flickering of images and lower power consumption. Finally, both display brightness and refresh rate need to be considered when evaluating the power consumption of displays.

Finally, according to Abdesslem *et al.* [30], network, sensors, utility, and display can greatly impact battery life. They provided a case study using a Nokia Na5 8GB smartphone, in which they demonstrated that when all components were off, the battery life lasted for 170.6 hours compared to 45.9, 21, 13.6, and 3.5 hours when the accelerometer, Bluetooth, microphone, and video camera, respectively, were active during the battery life duration, and 11 and 7.1 hours when GPS was used indoors versus outdoors.

2.4 Related Work

Analyzing and optimizing the energy efficiency of mobile devices is a well-researched topic. In order to attain this goal, researchers approached it from the following five angles: power profiling of hardware components using hardware-based methods, power profiling of hardware components using software-based methods, energy and power profiling of mobile applications, increasing the energy efficiency of hardware components, and increasing the energy efficiency of mobile applications.

2.4.1 Power Profiling of Hardware Components using Hardware-Based Methods

Using external power-measurement tools, Carroll and Heiser [32] analyzed the power consumption of smartphone components using a Data Acquisition system (DAQ) with an instrumented platform. They ran various benchmarks in order to accurately measure the power consumption of major components of a smartphone. Based on their analysis, the display, GSM module, graphics accelerator/driver, and backlight were the most power-consuming components.

Dong *et al.* [39] also relied on external measurement tools in order to power profile the graphical user interface on OLED displays at the pixel, image, and code levels. They achieved accuracy of 99, 90, and 95 percent, respectively. They built their energy models by measuring the power consumption of the display by collecting the current drawn from a USB interface of a DAQ.

Finally, Lajolo *et al.* [65, 64] propose coestimation-based power estimation for SoC design. They analyze different parts of the SoC using a system-level summation master.

2.4.2 Power Profiling of Hardware Components using Software-Based Methods

By only relying on software-based techniques for power profiling of device components, Maker *et al.* [68] provided a technique to improve online power modeling in smartphones. They conducted case studies where they profiled power consumption of different smartphone components such as Wi-Fi, GPS, and cellular radio by changing the battery management unit (BMU) sampling rate. As a result, they increased the accuracy of power consumption estimation of those devices.

Similarly, Jung *et al.* introduced DevScope [59], an online power-analysis tool for smartphone hardware components, which can accurately build the power models despite the high-interval update rate of the BMU. Sesame [40] is another accurate energy modeling tool that uses a smart battery interface to build accurate power models with low-interval estimation of power consumption.

In addition, many software energy-profiling tools utilized Nokia Energy Profiler to build their models. For instance, Perrucci *et al.* [81] conducted a large set of experiments on a Nokia device running Symbian OS 9.2. Their experiments aimed to measure the exact power consumption of all smartphone components while accounting for their different power states. They used Nokia Energy Profiler and verified their results by a multimeter. They determined no significant difference between the reported power consumption values from both. Likewise, Balasubramanian *et al.* [28]

used Nokia Energy Profiler to profile network activities of available network technologies. They developed a model for the energy consumption of network devices, which can account for devices' tail power. As a result, they were able to present a method that can reduce the tail power based on the RRC protocol.

Shye *et al.* [88] provided utilization-based power modeling, using Dalvik Application logger for Android mobile phones to collect usage data and periodically sending them to servers. They also estimate power consumption and the power breakdown using hardware components. Finally, they use a regression-based power model that uses high-level system measurements to estimate power consumption.

Finally, PowerBooter also focuses on power modeling [103] of Android devices. It targets the CPU, LCD, GPS, audio, Wi-Fi, and cellular communication components. The authors created a power model by designing a set of training applications that explicitly exercise all relevant system states. Using the built-in battery voltage sensors, they create a battery discharge for each individual component leading to the determination of each component state's power consumption. Then, they perform regression to derive the power model.

2.4.3 Energy and Power Profiling of Mobile Applications

Profiling the energy and power consumption of mobile devices focused on two different demographics: the first one focused on everyday users while the other focused on app developers.

Focusing on everyday users. Most of the tools in this category rely on collective information to build the energy consumption models. For instance, Carat [78] is a tool that sends coarse-grained statistics to servers residing in the cloud. The statistics sent include battery usage, running apps, the device model, and the operating system. Based on the data collected from the pool of users, the tool can profile the application's impact on battery life and send notifications to users such as the best configuration properties of their specific platform in order to increase battery life while running a specific application. Carat also notifies users about power-hungry apps and apps that contain energy bugs. Likewise, Wang *et al.* [99] used a collaborative approach to estimate the power consumption of mobile applications. They collected data from 120,000 Android users for about four weeks. The information collected contained battery traces and application switching events. Then, they used the data to build their power estimation model for mobile applications.

Focusing on app developers. A significant number of tools were developed enabling app developers to debug the energy efficiency of their applications. For example, Kansal *et al.* [62] introduced an energy profiler which lets developers make power-aware design choices and trade off between energy consumption and performance of their applications. Taking it a step further is Li *et al.* [67], who target mobile application developers by enabling them to perform source-line-level energy consumption profiling. They achieved this level of granularity by combining hardware-based power measurements in addition to program analysis and statistical modeling.

Focusing on function, process, and/or thread-level profiling. PowerScope [46] maps energy usage of mobile applications to processes and procedures. It requires hardware instrumentations in addition to kernel modification. Also, Etop [69] is a per-process energy profiler with a high resolution reaching a millisecond and targeting Linux-based systems. It can provide information in real time while continuously updating the energy profile of an entire application. In addition, pTop, developed by Do *et al.* [38], is a process-level power-profiling tool, running at the kernel level of Linux systems. The energy consumption of processes is based on the resource utilization of the latter. In addition, Pathak *et al.* [80] provide fine-grained power modeling for smartphones using system call tracing, which uses two types of models: utilization-based and nonutilization-based power behavior. This technique did not simply enable them to account for components' power based on their state, but also for the components' tail power, and then associate the values with the application responsible for the power consumption.

WattsOn [74] is another tool aimed at application developers. It allows them to focus on the energy efficiency of their code by mimicking the Windows Phone platform and estimating the app's energy consumption on the basis of empirically derived power models made available by either the smartphone manufacturer or mobile OS platform developers. Likewise, Eprof's [79] main goal is to capture and account for the power usage of the program entity by precisely accounting for the entitys effect on components' power state and accounting for the power consumed by the component even after the entity completed its functionality. The tool can be used by application developers in order to find the source code of energy bugs such as "wakelock bugs."

Finally, Wattch [31] is aimed at an even-lower level, focusing on enabling architects and developers of compilers to analyze and optimize microprocessor power dissipation at the architecture level.

2.4.4 Increasing the Energy Efficiency of Hardware Components

In order to optimize the system's energy efficiency, Intel Turbo Boost Technology 2.0 [16] was introduced, which controls the power and frequency of the CPU and processor graphics and utilizes a power predictor to determine the adjustments needed. The predictor's outputs are also exposed externally to software. Bellosa [29] went a step further, providing a thread-specific energy usage and using that information to control the CPU clock speed.

A smartphone's full potential can only be achieved by its ability to connect to the Internet. The common connection models are through 3G cellular data networks and Wi-Fi, in addition to the recent penetration of 4G LTE link, which is currently not supported in all areas. The smartphone usability is diverse and highly dependent on users demographics. Recent studies by Qian *et al.* [82] and Falaki *et al.* [45] revealed that the number of applications used varies from 10 to 90 per user, and the number of interactions per day varies from 10 to 200.

Due to the importance of Wi-Fi and the amount of energy it consumes, Kim *et al.* [63] introduced WiFisense, a Wi-Fi sensing system that maximizes the use of Wi-Fi access points while improving the energy efficiency through adaptive scan-triggering time intervals. Another Wi-Fi related project is Wi-Fi tethering, which refers to the use of a smartphone's Wi-Fi interface as a means to share its own Internet connection with other clients such as tablets, smartphones, or computers. DozyAP [51] is a system designed to put the Wi-Fi interface of a smartphone, which is acting as a mobile software access point, into timed and client-approved sleep mode in order to increase its energy efficiency. Similar to this work, we studied the potential of reducing energy consumption when using Wi-Fi during idle mode as opposed to their research which focus on Wi-Fi during active mode.

Finally, Qian *et al.* [43] characterizes the impact of 3G networks' operational state machine settings and provides insights into the present inefficiencies, which are due to the interplay between the devices applications and the state machine behavior. They then propose an optimal state machine setting that can, for instance, reduce the energy of streaming YouTube videos by up to 80%.

2.4.5 Increasing the Energy Efficiency of Mobile Applications

In order to increase the energy efficiency of mobile applications, researchers introduced API to be used by developers. For example, Senergy was developed by Kansal *et al.* [61]. It includes an API that can be used by developers of context-aware applications in order to enter latency, accuracy, and battery (LAB) requirements independent of sensors and inference algorithms. Then, Senergy attempts to meet developers' LAB requirements by adapting as the hardware changes. Another framework example is SystemSens [44], developed with the goal of monitoring usage of smartphones' research deployment. It has a client-server model where the apps on smartphones (clients) send periodical information to the server. A subset of the events sent are related to battery usage, screen status (on or off), service logs, and network traffic statistics such as Wi-Fi signal strength, just to name a few. Application developers can use the AIDL interface to be treated as a virtual sensor of the framework and thus collect context and power utilization data related to the application. This information can be collected and monitored by the application developers in order to increase the energy efficiency of their applications.

Another mean to increase the energy efficiency of application is through offloading parts of the applications execution from the mobile devices to other independent devices through the utilization of Wi-Fi connections. Gordon *et al.*[47] presents COMET (Code Offload by Migrating Execution Transparently), which can offload some of the execution of an unmodified multithreaded application to run on multiple machines. By allowing this type of offloading, the authors observed at least 1.7 times speedup as opposed to only running the application on the mobile device.

CloneCloud [34] was developed by Chun *et al.* The system partitions applications running on mobile devices and allows seamless offloading of parts of the applications execution from the mobile device onto device clones available on the cloud. Partitioning does not require modification of the mobile application. The system instead relies on offline static and dynamic profiling of the application. The static profiling step creates a database of possible partitions based on constraints (the application's method entry and exit points). Then, the dynamic profiler uses the database, which was made available by the static part, to profile the input executable on the mobile device as well as the cloud. Then, it creates the cost model for the set of possible partitions. Finally, the optimization solver finds the appropriate partition that meets the partitioning objective, which can either be to optimize the execution time or to optimize the energy efficiency.

2.5 Conclusion

In this chapter, we made a clear distinction between power and energy. Then, we explained the metrics affecting the power consumption of processors. Next, we provided a comprehensive energy overhead analysis of mobile devices, followed by discussion of related work.

CHAPTER 3: SOFTPOWERMON: A POWER-PROFILING TOOL FOR ALL ANDROID MOBILE DEVICES

3.1 Introduction

The use of smartphones and tablets is continuously on the rise. It is forecast that by 2015, the sales of these handheld devices will surpass the sales of notebook personal computers [24]. There are many reasons for the popularity of these devices. First of all, there are thousands of applications developed specifically for these handheld (mobile) devices, such as games, maps, video players, and reminders. Having thousands of these applications makes it easy for clients to rely on their devices to perform tasks that would have required a computer in the past. Second, the size of smartphones and tablets makes them portable and easy to carry. Third, Internet and data networks have become very accessible either through 3G or 4G LTE networks or Wi-Fi.

Smartphones and tablets are highly dependent on their battery life. What is the point of having a very powerful device with a short battery life? As a result, clients nowadays evaluate and compare devices and applications on the basis of their performance and energy efficiency. Therefore, these devices' energy efficiency is at the forefront of research topics related to mobile devices.

Needless to say, the competition in this field is very high. There are several operating systems in the market, such as Google's Android (currently the most popular OS), Apple's iOS, and Windows Mobile. The competition is not restricted to the platform but also extends to the mobile application market. Since energy efficiency is a key evaluation component, platform and application developers must power-profile their platforms and/or applications. It is not enough to simply understand how much power a device is consuming or to predict how much it might consume; there should be a deeper focus on understanding why a specific amount of power is consumed.

3.1.1 Motivation

Most of this research focuses on power modeling and battery life estimation, whereas we focus on understanding the impact of an application on the underlying platform. We developed a tool that allows microprofiling of CPU power consumption in order to give platform and application developers data on the effectiveness of power management of the processor and the impact of applications on the processor's power consumption. We focus on giving information on "why" a specific amount of power is consumed as opposed to "how" much.

3.1.2 Contribution

This chapter's contribution is as follows:

- We propose a methodology for measuring power consumption behavior of Android applications using a Software Power Monitor tool (SoftPowerMon). This tool aims to provide platform and software developers with a means of examining how their platform and/or Android applications can affect a device's battery life.
- We developed two versions of the tool: one running on a host computer and one running on an Android device.

- We presented two case studies, one using smartphones and one using a tablet. Through our case studies, we showed how applications can affect the platform's power consumption differently. We also showed how an application's behavior is similar when running on different devices and how the power consumption behavior differs from one platform to another.

3.1.3 Organization

The rest of this chapter is organized as follows: in Section 3.2, we list and explain power profiling techniques used to determine the energy efficiency of applications on Android devices. Section 3.3 introduces the Software Power Monitor tool and lists its advantages. We present case studies in Section 3.4, and in Section 3.5 we provide an overhead analysis of SoftPowerMon. In Section 5.4, we showcase a SoftPowerMon version that runs as an Android App. Finally, we present our conclusions in Section 3.7.

3.2 Power Profiling Techniques Used to Determine Energy Efficiency of Applications on Android Devices

Android is an open-source project published by Google since 2005. The Android mobile operating system is developed based on a modified Linux 2.6 Kernel. It uses native Linux libraries to manage and set policies of the Linux Power Manager. Energy conservation can be achieved by changing the unused components' power states during idle time, changing the power performance states of the CPU based on the CPU's load, and reducing/eliminating unnecessary system wake-ups.

3.2.1 Processor Idle Sleep States

Modern processors attempt to reduce their power consumption by supporting different idle states known as C-States. The power savings is achieved by turning off the CPU's unused architecture blocks. The C-States supported by a CPU are dependent on the type of CPU. Regardless of the CPU type, the deeper the idle sleep state, the greater the power savings, and the more components in the CPU are switched off [17]. On the other hand, the deeper the idle state, the longer it takes for the processor to move from that idle state to an active state.

There are two types of idle states: core idle states and package-level idle states. In multicore devices, each core can have its own independent idle state. However, regarding the package idle states, a package cannot enter deeper idle states unless all the cores in the system agreed to enter the same level of idle state or a deeper one.

An application's energy efficiency can be measured by the percentage of time it allows the processor to enter an idle state. A Linux kernel gives us enough information to be able to determine the processor states. The kernel keeps track of all the idle states supported by each processor, the idle state's corresponding name and description, its exit latency in microseconds, time spent in that state in microseconds, and power consumed in milliwatts. All this information can be found under */sys/devices/system/cpu/cpuXX/cpuidle*, where XX represents the core number. Finally, the time spent in active state for each core can be determined as shown in Equation 3.1 where D is the total test duration, n is the number of core idle sleep states, and $Idle_{State_n}$ is the time a core spent in idle state n.

$$Active = D - \sum_{i=1}^{n} Idle_{State_n} \tag{3.1}$$

Power Performance States

Power performance states known as P-states are the processor's frequency and/or voltage. Each processor has a set of supported frequencies. The processor's frequency can impact both performance and power consumption. There is a direct relationship between performance and power consumption. We reduce the processor's performance by lowering the processor's frequency, and thus we decrease its power consumption, and vice versa.

The Linux kernel can have different policies for manipulating the processor's frequency. These policies can be enforced through the platform's "governor." The platform may support several governors. Typically, the governor's policy for P-states is based on the CPU load, where in the event of load changes, the CPU frequency changes as well. Governors can also be set to "powersave," where the lowest frequency is selected at all times, or set to "performance," where the highest frequency is set at all times. A user may set the frequency by accessing the directory */sys/devices/system/cpu/cpu0/cpufreq/*. In terms of power profiling of platforms or applications, we are interested in the time spent in each frequency. This information is available at */sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state*. This information will let us determine the pattern of power consumption of the CPU when it was in active state.

Dynatick (Tickless Operations)

Traditionally, the Linux kernel used a global timer for timekeeping purposes. This timer—also known as "timer tick"—causes timed interrupts. Because these interrupts don't account for the CPU's idle sleep states, they will cause the CPU to switch to active state in the event that the timer was triggerd when the CPU was in idle sleep state. The kernel has long moved from "timer tick" mode to dynamic tick "Dynatick" mode instead, where timer interrupts occur only when needed [90]. These ticks are particularly important for power profiling of a platform because they can have a direct impact on the duration the CPU can remain in the idle residency state. By accessing */proc/time_stats*, the system reports the Dynaticks in addition to total number of events, along with the average events per second. This information is particularly important for the kernel, operating system, and platform developers.

System Interrupts

Linux Kernel also keeps track of the resources that woke the system up. It keeps track of the number of interrupts per CPU. It detects and records the internal interrupts, such as the nonmaskable interrupts (NMI), local timer interrupts (LOC), TLB flush interrupts (TLB), rescheduling interrupts (RES), and remote function call interrupts (CAL). It also records the external interrupts for I/O devices. The system interrupt data can be collected from */proc/interrupts*. The fewer number of interrupts results in fewer wake-ups, leading to less power consumption.

3.3 Software Power Monitor Tool (SoftPowerMon)

We developed a power analysis tool using Python that can be used on any Android platform to perform debugging of the energy efficiency of Android applications and processors with negligible overhead. SoftPowerMon requires that Android Debug Bridge (adb) [37] be installed on a host computer. The tool runs on the host computer and can access the platform under testing using a USB connection. Only rooted devices may be profiled. Finally, the tool provides an output file containing all of the raw data collected in addition to post-processing data.

- **Processor Idle Sleep States Test:** Using SoftPowerMon, we can collect the time spent in each idle sleep state for each core. In addition, we can collect the usage value for each idle state. This usage value represents the number of transitions to a specific idle state. In our implementation, we considered that some types of processors can turn one or more cores offline during the test duration. As a result, we continuously poll the list of online cores. Finally, during the post-processing step, we calculate the percentage of time the core spent offline based on the list of online cores, in addition to duration in each idle sleep state. Finally, we use the total offline duration and total time spent in each idle sleep state to calculate the total active duration.

- **Power Performance States Test:** In order to determine the time spent in each frequency, we take a snapshot of the P-states time_in_state for each frequency at the beginning and end of the test. The difference between the end values and the beginning values represents the time spent in each frequency.

- **Online Test:** For the online test, we continuously poll the list of online cores from the system along with the timestamp.

- **Dynamic Tick Events Test:** To collect the dynamic tick events, we simply prompt the system to collect those data during a specific period of time. Then we extracted that information to the tool user.

- **Other Tests:** We also created a test for collecting system interrupts, and another test to collect some statistics about the system, such as the number of cores, the available idle sleep states with their corresponding wakeup latency, and the power-performance states supported, along with statistical information such as the maximum and minimum power-performance states.

In addition to proving power profiling mean to the users, we also include a means for users to control the device's power. Using SoftPowerMon, users can change the device governor in addition to changing the power performing frequency of the device.

The combination of all these data can give a clear view about the energy efficiency of a platform or application and keeps the focus from how much power is consumed to why this much power is consumed.

3.3.1 Advantages of SoftPowerMon

The advantages of using SoftPowerMon are as follows:

Table 3.1: Benchmark description.

Benchmark	Description
Google V8	It contains pure JavaScript benchmarks for OS kernel simulation, one-way constraint solver, Encryption and decryption, classic Scheme, and regular expression benchmark [12]
NenaMark2	It is an independent graphics benchmark made to test the performance of embedded GPU systems. It uses the industry standard OpenGL—ES 2.0 API to push GPUs to their limit in a setup that resembles that of a game. [25]
Sunspider	Browser JavaScript benchmark workload used to cross browser to test JavaScript performance [19]
GL	It is a performance-testing utility that measures different graphics and computation capabilities of a mobile device. The tests focus on graphic resources, measuring the quality and performance of the underlying OpenGL ES 2.x implementation. The benchmark contains high-level 3D animation [56]
YouTube	Played Elephant Dreams movie [41]
Idle	Screen on and nothing running on the background

- SoftPowerMon can answer the question of *why* a specific amount of power is consumed as opposed to *how much*.
- Application developers may use it in order to observe power consumption behavior of the application.
- Application developers can use it to examine the impact of their application on the platform in terms of wakeups.
- It may be used as a first-level triage without imposing extra overhead to the power consumption of the platform under test.
- It does not require any flashing of the kernel.
- It can collect data on any type of processor. By using SoftPowerMon, a platform manufacturer can strictly power profile the processor without taking into consideration any other device component of the platform. This enables developers to perform an apples-to-apples comparison of power consumption of an application running on different platforms. Thus, they can determine the energy efficiency of one processor compared to another.
- Platform developers can use it in order to debug the energy efficiency of their processors specifically when the platform is in idle (meaning no active workload). Since SoftPowerMon does not impose an overhead, then platform developers can use it to collect power behavior of the processor and observe the percentage of time the processor spent in active state and in available idle sleep states. If they observe high active duration or that the processor is not entering deep C-states or the processor is in high frequency (as opposed to low frequency), then they can determine that the power management of the processor is not effective and they can dig deeper in order to resolve the issue.

3.4 Case Studies

We performed two case studies: one using smartphones and the other using a tablet. The first case study used two smartphones: a Samsung S3 and a Samsung Galaxy Nexus 3. The second case study uses a Motorola XOOM tablet. For all the devices, we ran the Android benchmarks as listed in Table 3.1 where we collected the power performance and idle sleep states in addition to the dynamic ticks.

3.4.1 Smartphones: Samsung S3 and Samsung Galaxy Nexus 3

The first smartphone device used was a rooted Samsung S3 smartphone device, model number GT-19300, running Android version 4.0.4 Ice Cream Sandwich with Linux Kernel 3.0.15. The smartphone features an Exynox Quad core (Exynox 4412) also known as a Quad ARM Cortex-A9 core. This type of processor can turn one or more cores offline.

This processor's power modes are as follows [33]:

- **ARM Clock Gating (WFI)** is the first level of idle sleep state also known as state0. When the processor is in this state, most of the processor's clocks are disabled while its logic is still powered up. The exit latency is 1 us.
- **ARM Power Down** is a semi-idle state corresponding to state1 where the processor is powered down while the caches remain powered up and maintaining their state. The exit latency is 300 us.

The second device used was a rooted Samsung Galaxy Nexus 3 running Android 4.1.1 Ice Cream Sandwich. The device features an ARM dual core 1.2 GHz Cortex-A9. This processor's power modes are as follows:

- **C1 (WFI)** is mapped to state0. It is similar to the WFI state explained for Samsung S3. Its exit latency is 4 us.

- **C2 (CPUs OFF, MPU + CORE INA)** is state1, where the CPU is off, the Memory Protection Unit (MPU) is on to protect critical data, and the core is inactive. The exit latency for this state is 1100 us.
- **C3 (CPUs OFF, MPU + CORE CSWR)** is state2 and similar to state1 with the exception that the core is in Closed Switched with Retention mode. The exit latency for this state is 1200 us.
- **C4 (CPUs OFF, MPU CSWR + CORE OSWR)** is state3, similar to state1 with the exception that the core is in Open Switched Retention mode. The exit latency is 1500 us.

We collected the idle states and power-performance states on both devices for our entire list of benchmarks.

Figure 3.1 displays the data collected for Samsung S3 Idle states' residency values. By examining the graph, you will notice that, for few benchmarks, the data collected do not add up to 100 percent. During those cases, the core was neither in Active state nor in WFI state, but offline instead. Figure 3.1 also shows that core1 remained for over 80 percent in Active state, whereas core2 was offline during the Google V8 benchmark. These results show that Google V8 does not run efficiently on the device because, despite trying to save power by switching the status of core2 to offline, core1 remained in Active state most of the time and thus prevented the platform from going into the low-package idle state.

The idle states of Nexus are displayed in Figure 3.2. Google V8 behaved similarly as its behavior on S3. Even though core1 was not turned offline, there was no idle-state balance between core0 and core1 (Note: Nexus does not support switching cores to offline modes). Figure 3.2 also reveals that the Sunspider benchmark had a higher percentage of active states compared to the other benchmarks.

Figures 3.3 and 3.4 displays the number of idle states transitions to WFI per second for Samsung S3 and Nexus, respectively. By comparing S3 and Nexus, it is clear that the number of transitions for S3 is much higher than Nexus. This information is helpful for platform developers because an overhead is incurred when the processor keeps switching states. Therefore, platform developers must understand whether this continuous switching of states is due to an optimized platform that switches to idle states whenever possible or whether there are unnecessary interrupts that switch the CPU state from idle to active on a continuous basis leading to a cycle of switching between active and idle.

Figure 3.5 displays the percentage of time spent in each power-performance state for Samsung S3 and Nexus for all scenarios. Google V8 power consumption is the highest because both phones remained in the maximum power-performance frequency during the majority of the test duration. Another valuable observation for platform developers is the power-performance states of S3 during the idle scenario compared to Nexus. S3 stayed for 17.57 percent of the time in a power-performance state higher than its minimum frequency, whereas Nexus spent only 3.7 percent. During idle time, the platform is required to consume the lowest possible amount of power. As a result, this gap must be reduced to increase S3's energy efficiency.

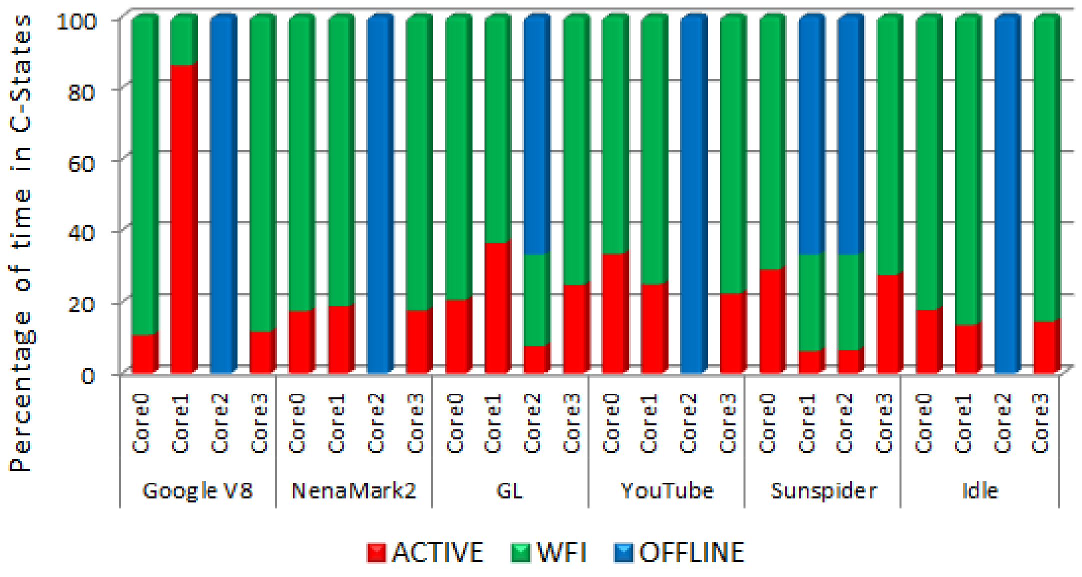


Figure 3.1: Comparing percentage of time in idle states residency for Samsung S3.

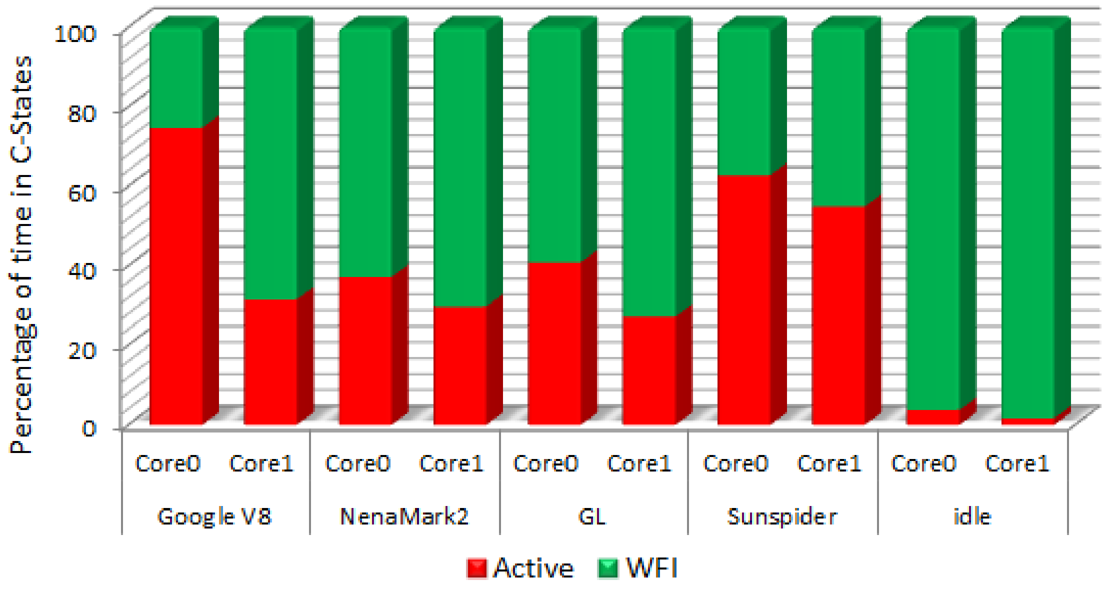


Figure 3.2: Comparing percentage of time in idle states residency for Nexus 3.

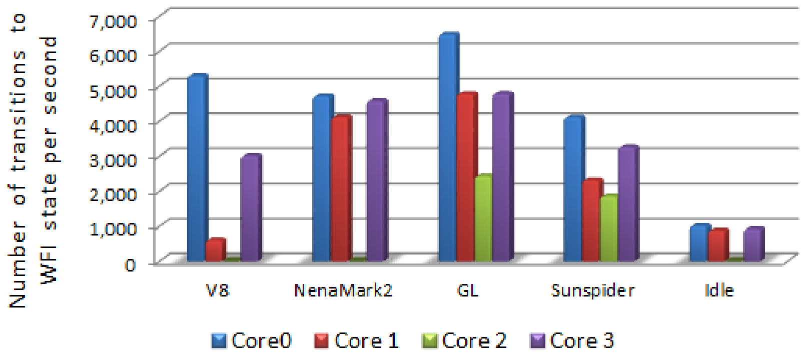


Figure 3.3: Comparing number of idle states transitions to WFI per second for Samsung S3.

Table 3.2: Benchmarks DynTicks events per second on Samsung S3.

Benchmark	DynTicks Events Per Second
V8	712.902
NenaMark2	502.227
GL	546.378
Sunspider	767.662

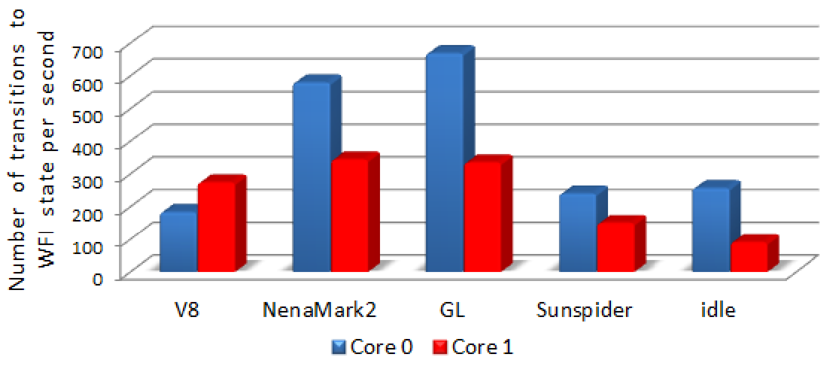


Figure 3.4: Comparing number of idle states transitions to WFI per second for Nexus 3.

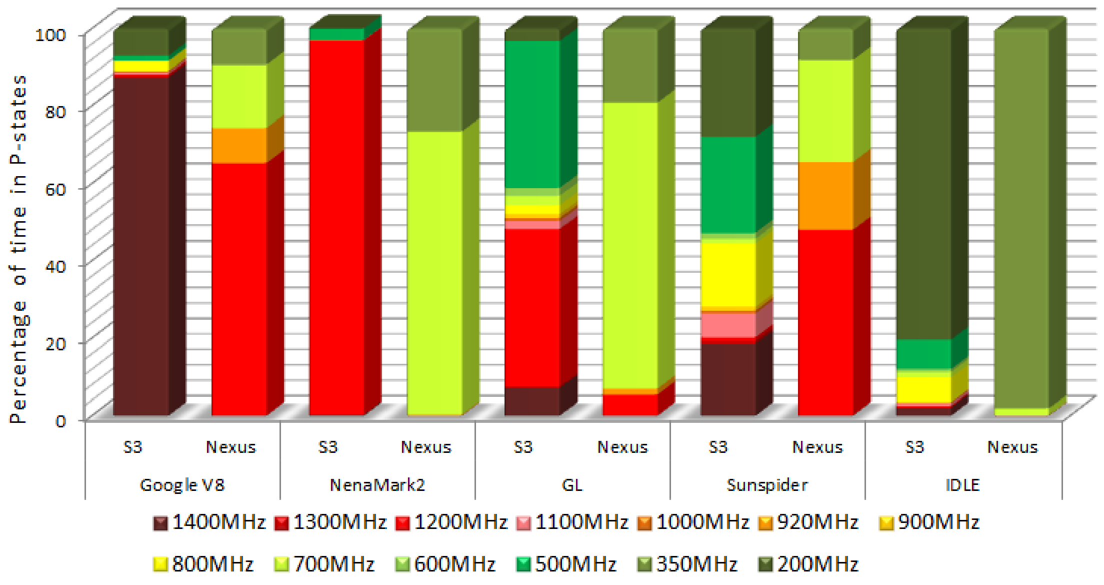


Figure 3.5: Comparing percentage of time in power performance states residency for Samsung S3 and Nexus 3.

3.4.2 Tablet: Motorola XOOM

The device used is a rooted Motorola XOOM tablet running Android 4.1.1 Ice Cream Sandwich with Linux Kernel 2.6.39.4. The XOOM tablet features a NVIDIA Tegra 2 SoC integrated with dual-core ARM Cortex-A9. This processor offers two idle-state modes:

- **CPU Flow Controlled mode (LP3)** is the first level of idle sleep state (state0). It is characterized by having the CPU turn off all the components except the memory units. This state's exit latency is 10 us.
- **CPU Power Gate (LP2)** is state1 idle state and is characterized by a complete shutdown. This state's exit latency is 1500 us.

For this case study, we followed the same testing methodology as we did for the smartphones. Figure 3.6 displays percentage of time spent in each idle states residency for the tablet. It is evident that both cores remained in a high percentage of active states during the YouTube case scenario. We also observed that core0 remained active for a longer duration compared to core1 during Google V8 and Sunsipider. In other words, there is a significant gap between active state duration among both cores. This implies that the application's load is not balanced among cores. This imbalance can negatively impact the percentage of time spent in package-level idle state residency. Because the package can only enter idle states when both cores are idle, this imbalance leads to longer duration of package active state. This explains the reason behind core0's higher numbers of transitions from active state to idle states compared to core 1, as displayed in Figure 3.7. In addition, since the load is concentrated on one core instead of two and since the frequency is adapted based on the load of the CPU, then this fact consequently explains the data collected for P-states, as shown in Figure 3.8, where the CPU remained in the max CPU frequency most of the time while running the two discussed benchmarks.

Unlike Google V8 and Sunspider, Nenamark2 and GL benchmarks had a balanced percentage of time in active state between core0 and core1. In addition, the CPU spent only around 50 percent of the time in the max frequency and between 31 and 24 percent of the time in the min frequency.

Finally, we showed how, by using SoftPowerMon, we gained insight on the benchmarks' power behavior at the micro level without the need for any sophisticated tool. We also showed that across the devices, the benchmarks had similar behaviors patterns. This is an important characteristic because it lets application developers test only their applications on a small subset of devices.

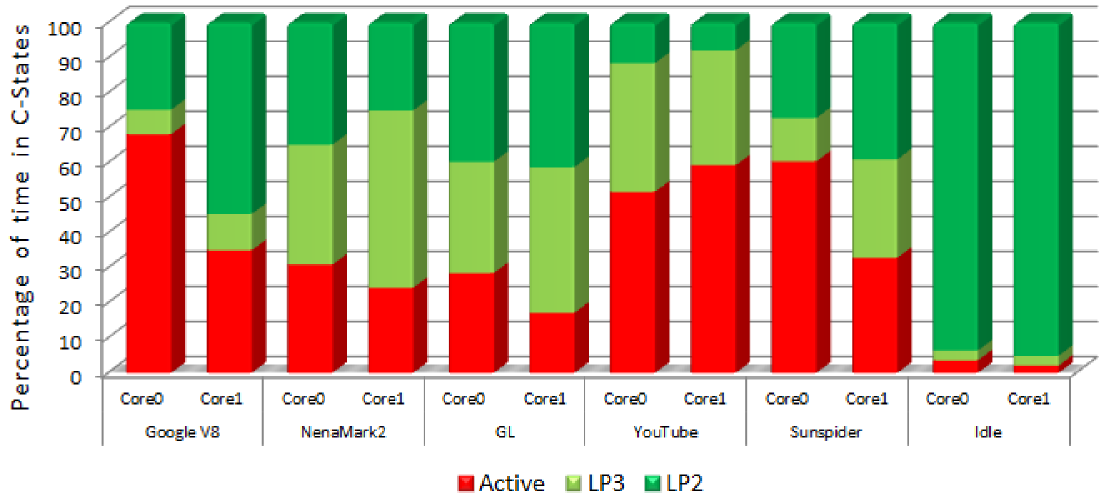


Figure 3.6: Percentage of time in idle states residency for Motorola XOOM.

3.5 Performance Analysis of SoftPowerMon

One main goal of our SoftPowerMon tool is to be able to collect power consumption behavior of Android applications with low overhead. We compared our tool to similar tools for profiling power consumption of Android applications. The first tool we examined is PowerTutor [101], which provides information such as energy usage over time of different phone components, including LCD, Wi-Fi, CPU, and 3G. The second

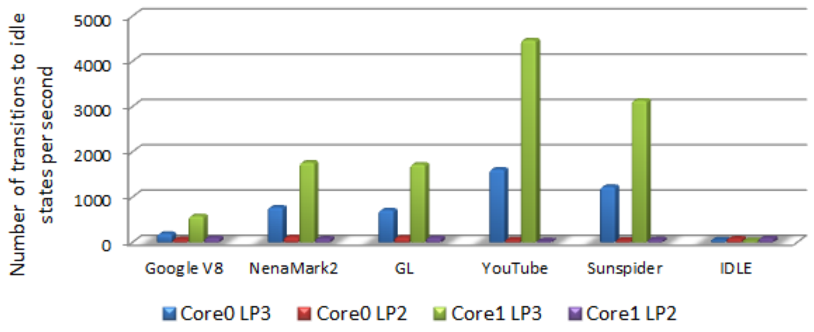


Figure 3.7: Comparing number of transitions to idle states per second for Motorola XOOM.

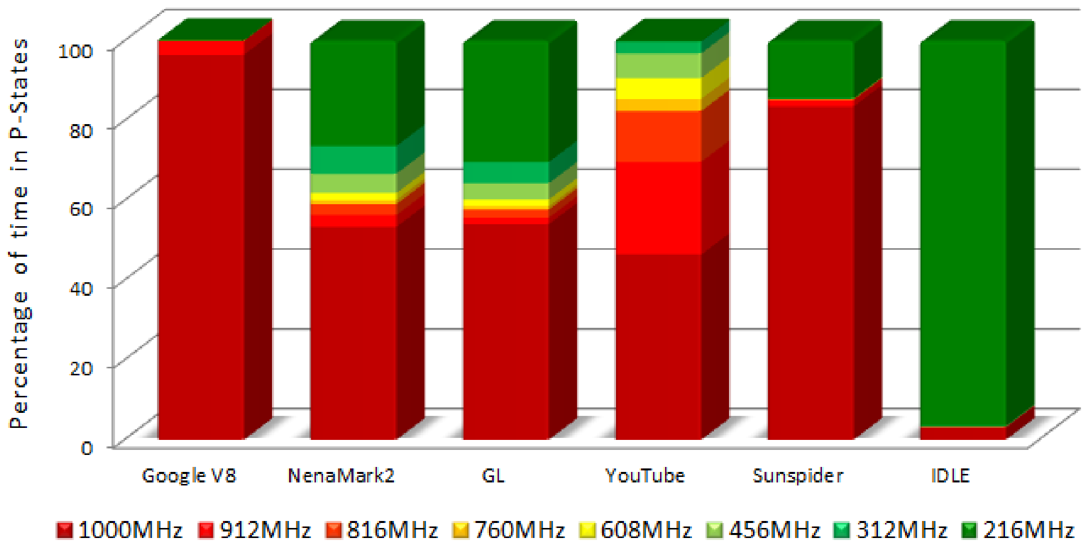


Figure 3.8: Comparing percentage of time in power performance states for Motorola XOOM.

tool we examined is System Panel App / Task Manager Pro [6], which displays CPU, memory, and network activities in addition to battery usage of the device.

In order to evaluate our tool's performance overhead, we randomly selected NenaMark2 from our list of benchmarks. We collected the data using SoftPowerMon on Samsung S3 while following three different scenarios: (1) running the benchmark alone; (2) running the benchmark along with running PowerTutor in the background; or (3) running the benchmark along with System App/Task manager Pro running in the background. Figure 3.9 represents the results of idle states residency percentages collected during the three scenarios, where SPMon, Ptutor, and SysPan are associated with scenarios 1, 2, and 3, respectively.

Based on the results, it is evident that, using SoftPowerMon without any other profiling tool, the device remained in active states for less time when compared to running it while other profiling tools were collecting data. Table 3.3 represents the performance states residency results for the same above scenarios. By comparing the results, we noticed that by solely using SoftPowerMon, the device could run during 5.96 percent of the time in the low P-state frequency of 500 MHz, whereas it spent 0 percent of time in the same frequency during the other two scenarios. In addition, by solely using SoftPowerMon, the device remained around 6 percent less time in the high P-state frequency of 1200 MHz when compared to the other two scenarios.

Another strategy to evaluate the overhead of collecting data via SoftPowerMon was to observe the impact of SoftPowerMon on the power and CPU utilization data collected using PowerTutor and SystemPanel. We noticed that there were no variations in the data collected via the latter two tools when we compared the results obtained while running SoftPowerMon and without running it.

Finally, PowerTutor and SystemPanel can collect more information than SoftPowerMon. However, if a user's intent is to just collect the impact of an application on the power utilization of the CPU and look at its direct impact on the idle state residency and power frequency, then SoftPowerMon is the best choice. It can provide

Frequency	SoftPowerMon	PowerTutor	SystemPanelPro
1400 MHz	0.20	0.27	0.64
1200 MHz	93.71	99.73	99.36
1100 MHz	0.06	0	0
700 MHz	0.05	0	0
500 MHz	5.96	0	0

Table 3.3: Comparing percentage in P-states during NenaMark2 benchmark

all this data to the user with minimal overhead. Knowing the percentage of CPU active state and its frequency can give a microlevel perspective on why an application is consuming a specific amount of power.

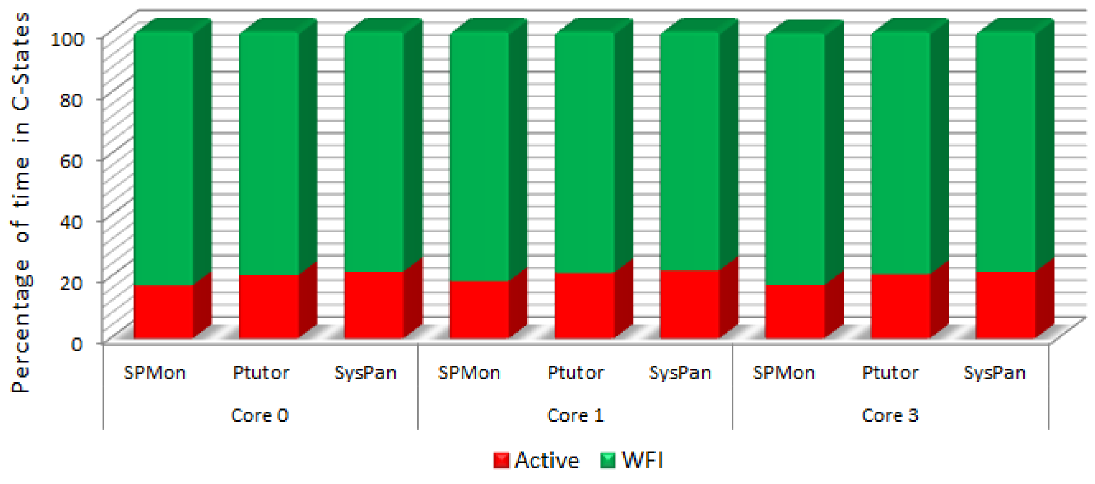


Figure 3.9: Comparing percentage in C-States when running SoftPowerMon, PowerTutor, and SystemPanelPro during NenaMark2 benchmark.

3.6 SoftPowerMon - The Android App

We also developed a version of SoftPowerMon that runs on an actual Android device instead of a host computer. There are distinct advantages for each version of SoftPowerMon. The advantage of having an Android-based version to run on the device is thereby eliminating the need of having the device rooted. On the other hand, since every application running on a device poses a power usage overhead, having a version that runs on a host platform instead ensures that the power usage overhead is close to null.

3.6.1 Description of SoftPowerMon - The Android App

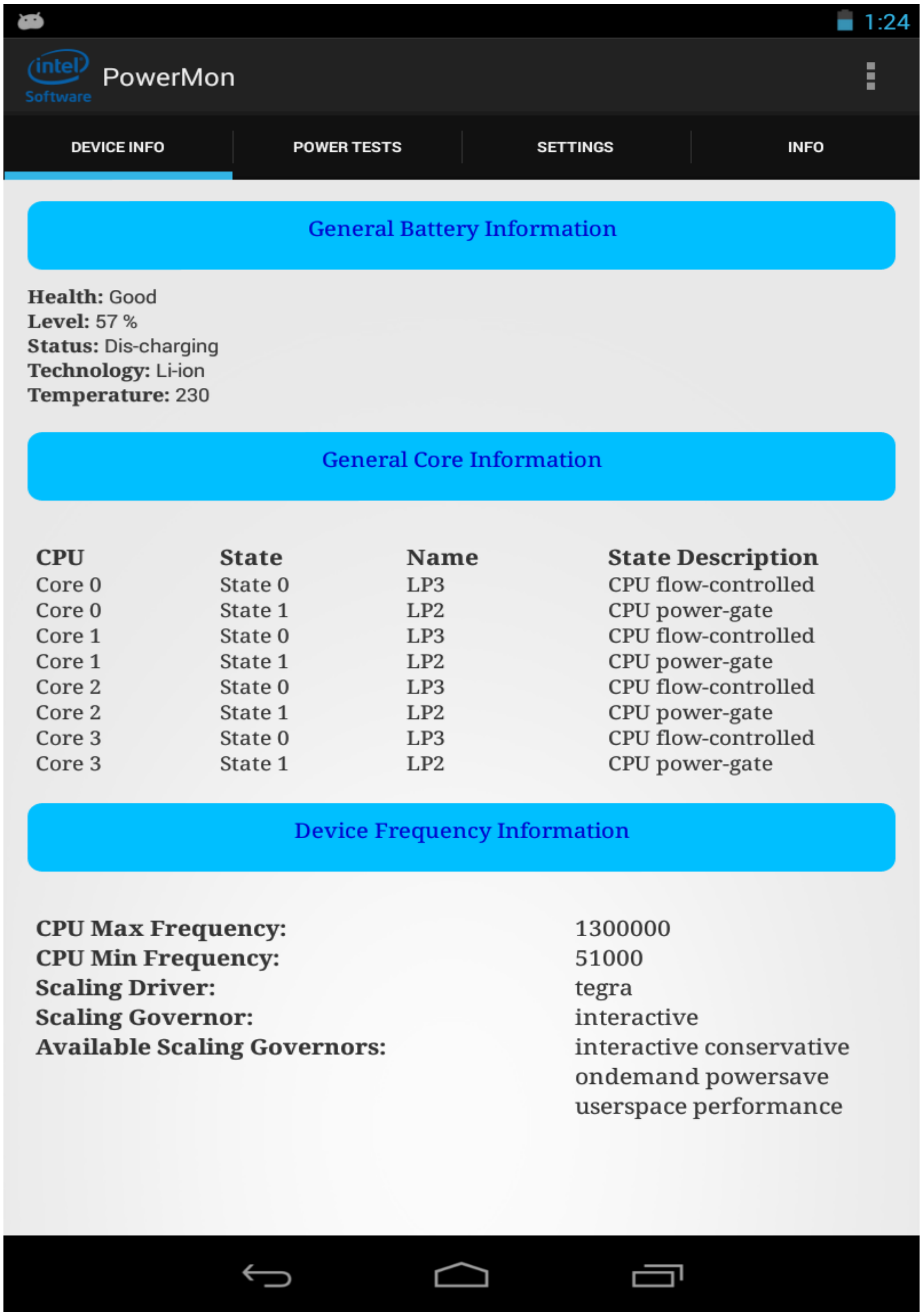
The Android app version of SoftPowerMon contains the following sections:

- **Device Info:** The Device Info section gives general battery information, core information, and device frequency information, as shown in Figure 3.10.
- **Power Tests:** The Power Tests section lets the user select a test scenario to run or run all tests. The user can also select the test delay time and duration, as shown in Figure 3.11.

  Once a test is completed, the user can save the test results and view them in a graph. Figures 3.12, 3.13, and 3.14 display the tabular results for the C-states, P-states, and Core Online results, respectively. Figures 3.15 and 3.16 display the graphical results of C-states and P-states, respectively.
- **Settings:** The Settings section displays all the available governors and frequencies and lets the users change them based on their preferences as shown in Figure 3.17.
- **Info:** The Info section explains all the tests.
- **History:** The History section lets the user view the results of tests previously collected on the device.

3.7 Conclusion

In this chapter, we list and explain power profiling techniques used to determine energy efficiency of applications on Android devices. We developed SoftPowerMon, a



General Battery Information

Health: Good
Level: 57 %
Status: Dis-charging
Technology: Li-ion
Temperature: 230

General Core Information

CPU	State	Name	State Description
Core 0	State 0	LP3	CPU flow-controlled
Core 0	State 1	LP2	CPU power-gate
Core 1	State 0	LP3	CPU flow-controlled
Core 1	State 1	LP2	CPU power-gate
Core 2	State 0	LP3	CPU flow-controlled
Core 2	State 1	LP2	CPU power-gate
Core 3	State 0	LP3	CPU flow-controlled
Core 3	State 1	LP2	CPU power-gate

Device Frequency Information

CPU Max Frequency:	1300000
CPU Min Frequency:	51000
Scaling Driver:	tegra
Scaling Governor:	interactive
Available Scaling Governors:	interactive conservative ondemand powersave userspace performance

Figure 3.10: Device info screenshot.

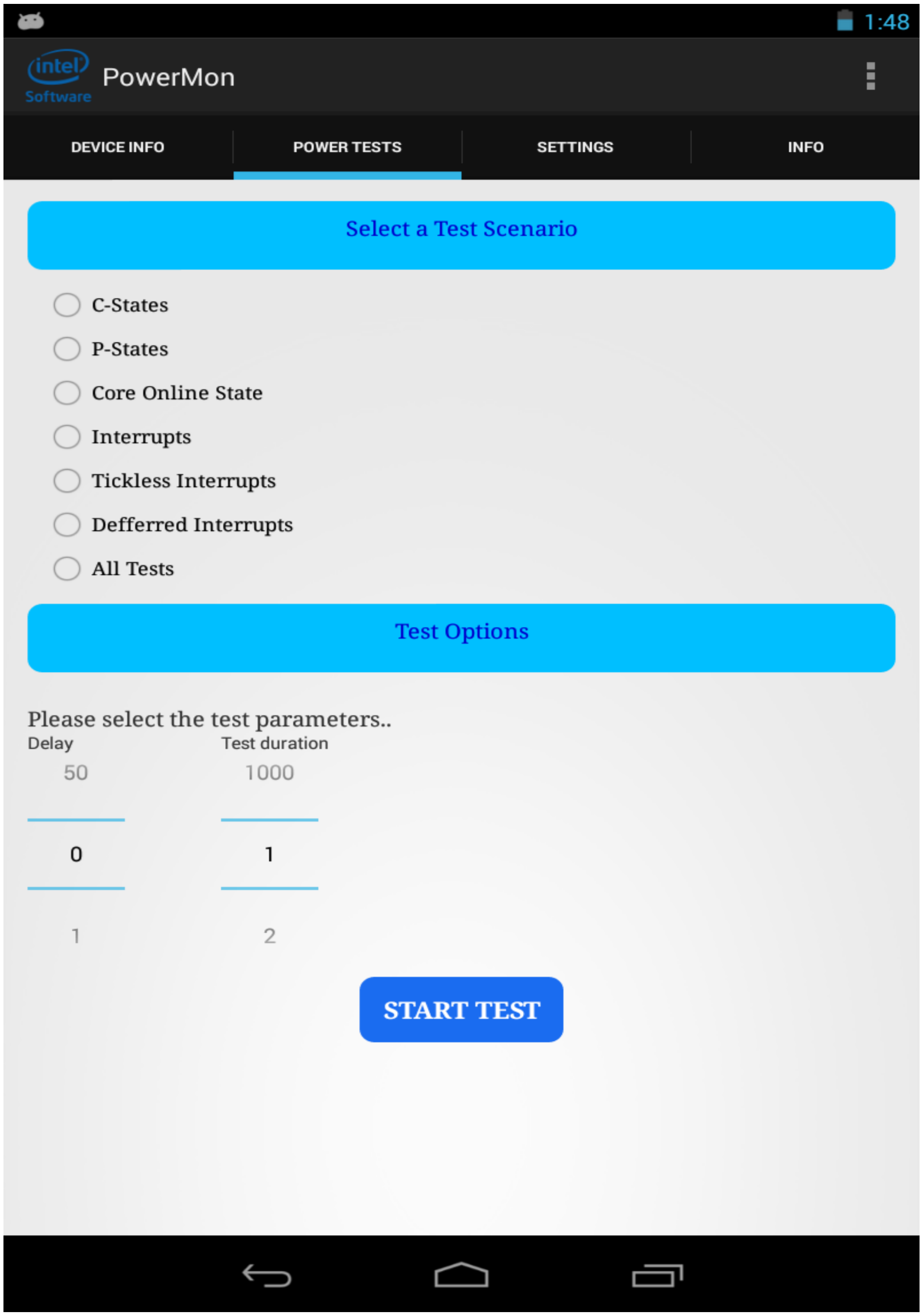
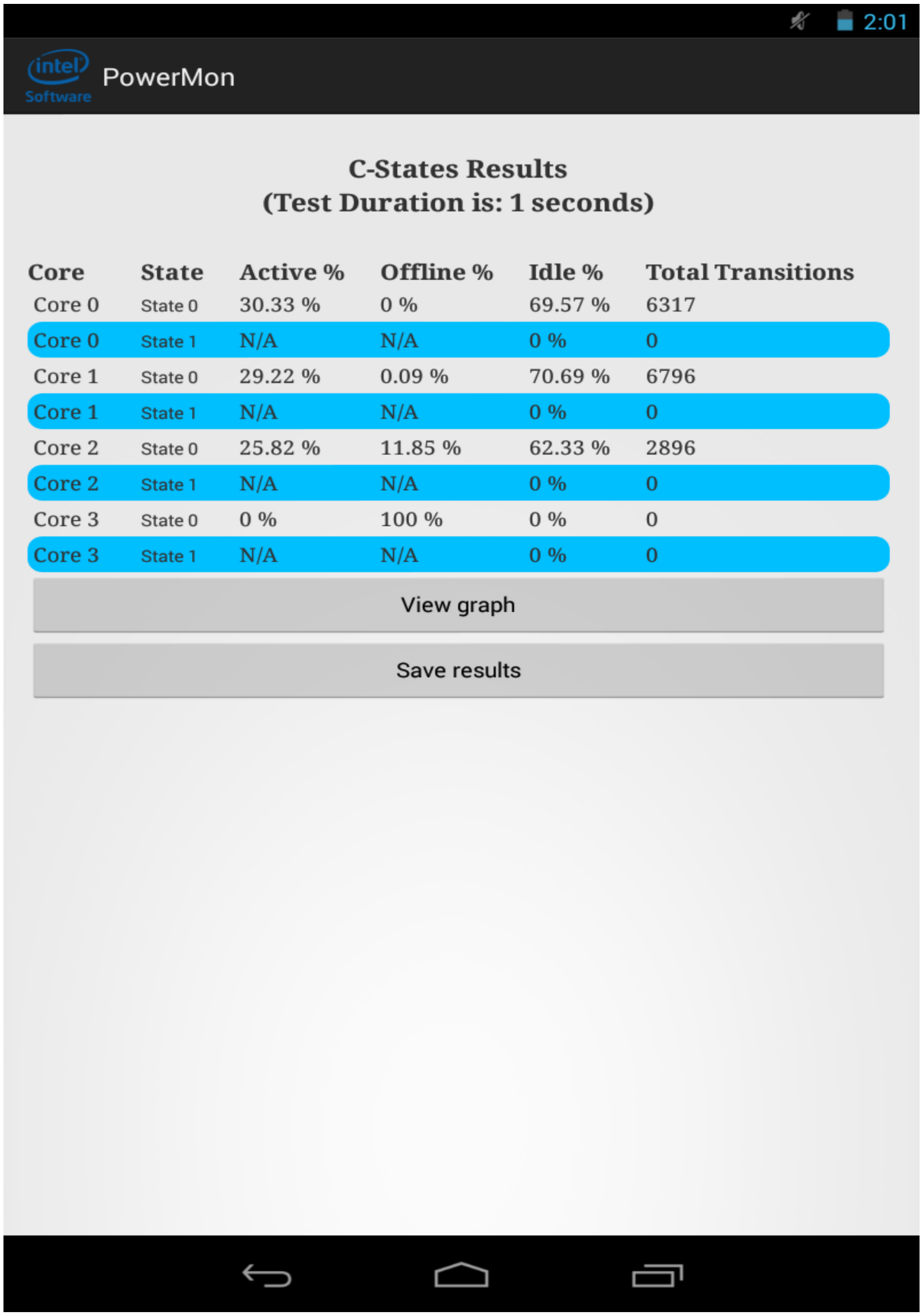


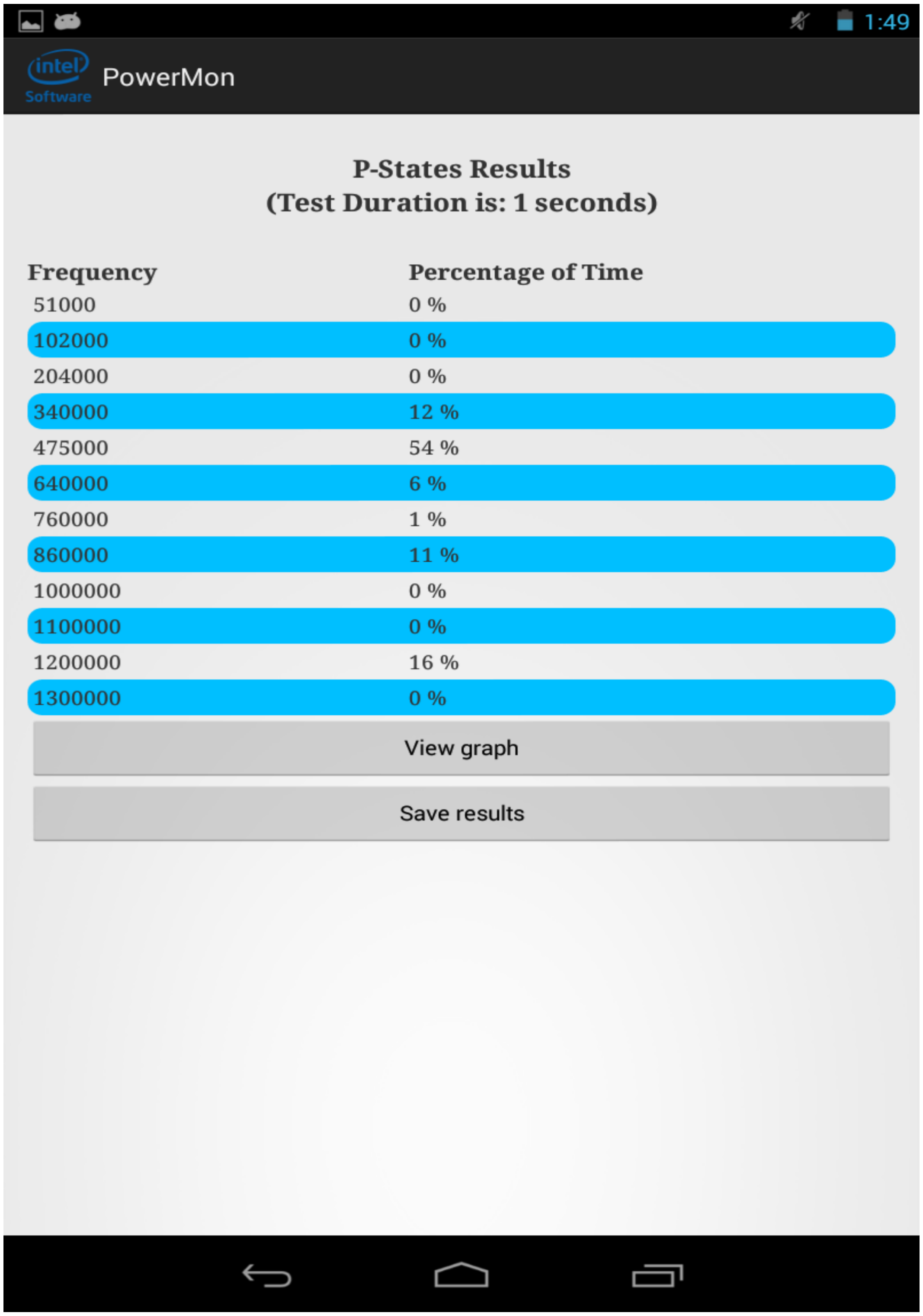
Figure 3.11: Power tests screenshot.



C-States Results
(Test Duration is: 1 seconds)

Core	State	Active %	Offline %	Idle %	Total Transitions
Core 0	State 0	30.33 %	0 %	69.57 %	6317
Core 0	State 1	N/A	N/A	0 %	0
Core 1	State 0	29.22 %	0.09 %	70.69 %	6796
Core 1	State 1	N/A	N/A	0 %	0
Core 2	State 0	25.82 %	11.85 %	62.33 %	2896
Core 2	State 1	N/A	N/A	0 %	0
Core 3	State 0	0 %	100 %	0 %	0
Core 3	State 1	N/A	N/A	0 %	0

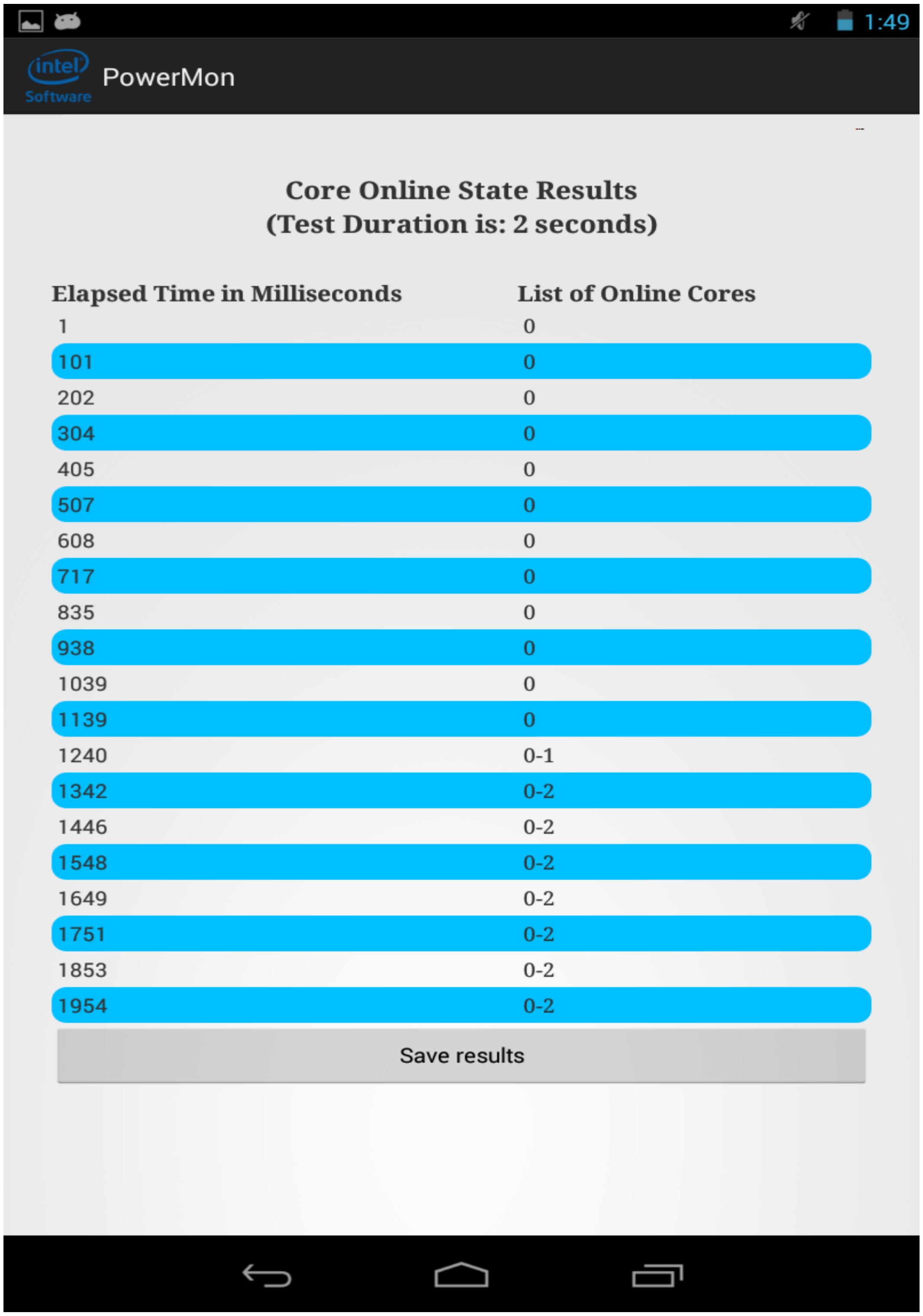
Figure 3.12: C-states tests results.



P-States Results
(Test Duration is: 1 seconds)

Frequency	Percentage of Time
51000	0 %
102000	0 %
204000	0 %
340000	12 %
475000	54 %
640000	6 %
760000	1 %
860000	11 %
1000000	0 %
1100000	0 %
1200000	16 %
1300000	0 %

Figure 3.13: P-states tests results.



Core Online State Results
(Test Duration is: 2 seconds)

Elapsed Time in Milliseconds	List of Online Cores
1	0
101	0
202	0
304	0
405	0
507	0
608	0
717	0
835	0
938	0
1039	0
1139	0
1240	0-1
1342	0-2
1446	0-2
1548	0-2
1649	0-2
1751	0-2
1853	0-2
1954	0-2

Figure 3.14: Core online test results screenshot.

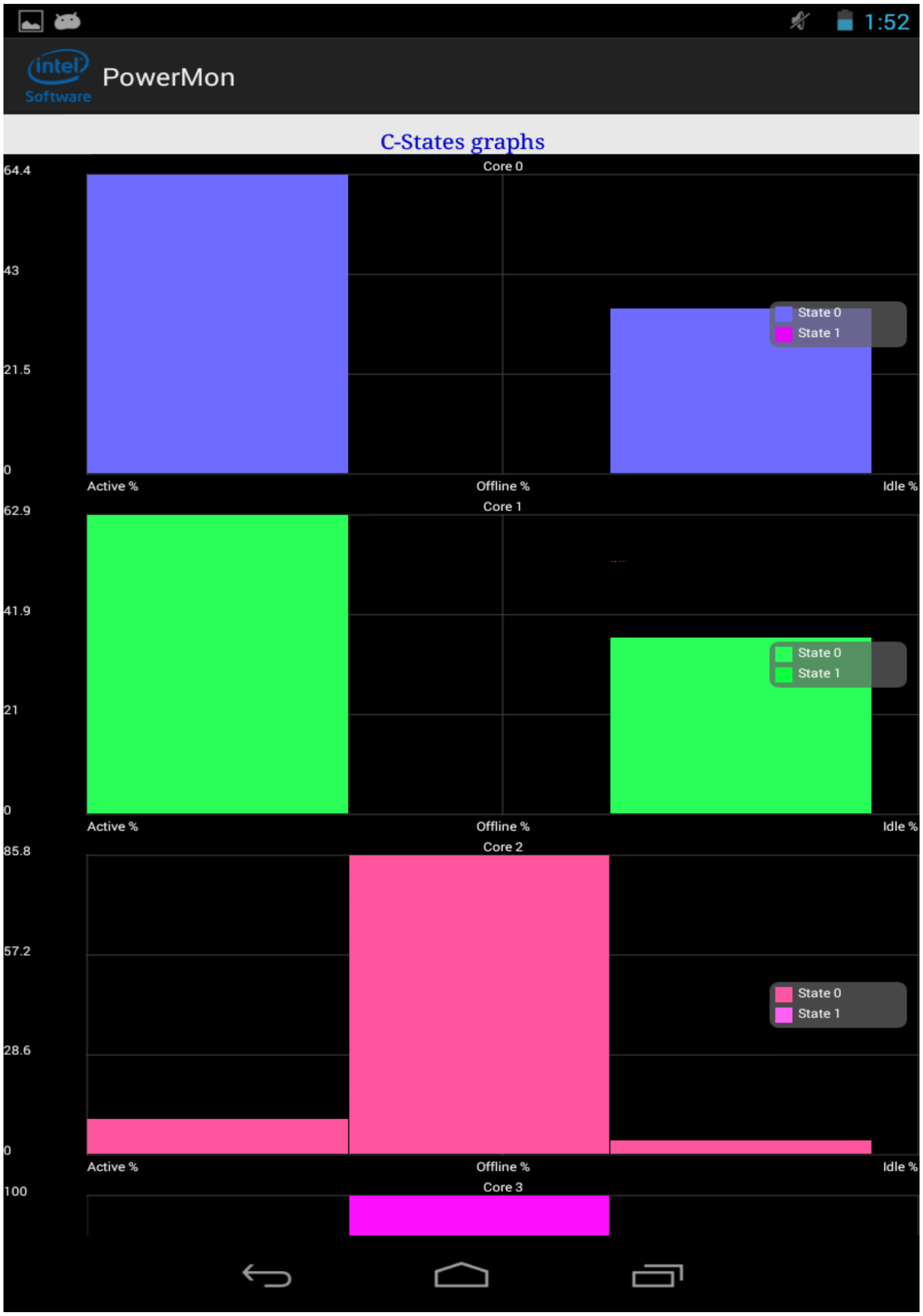


Figure 3.15: C-states results graph.

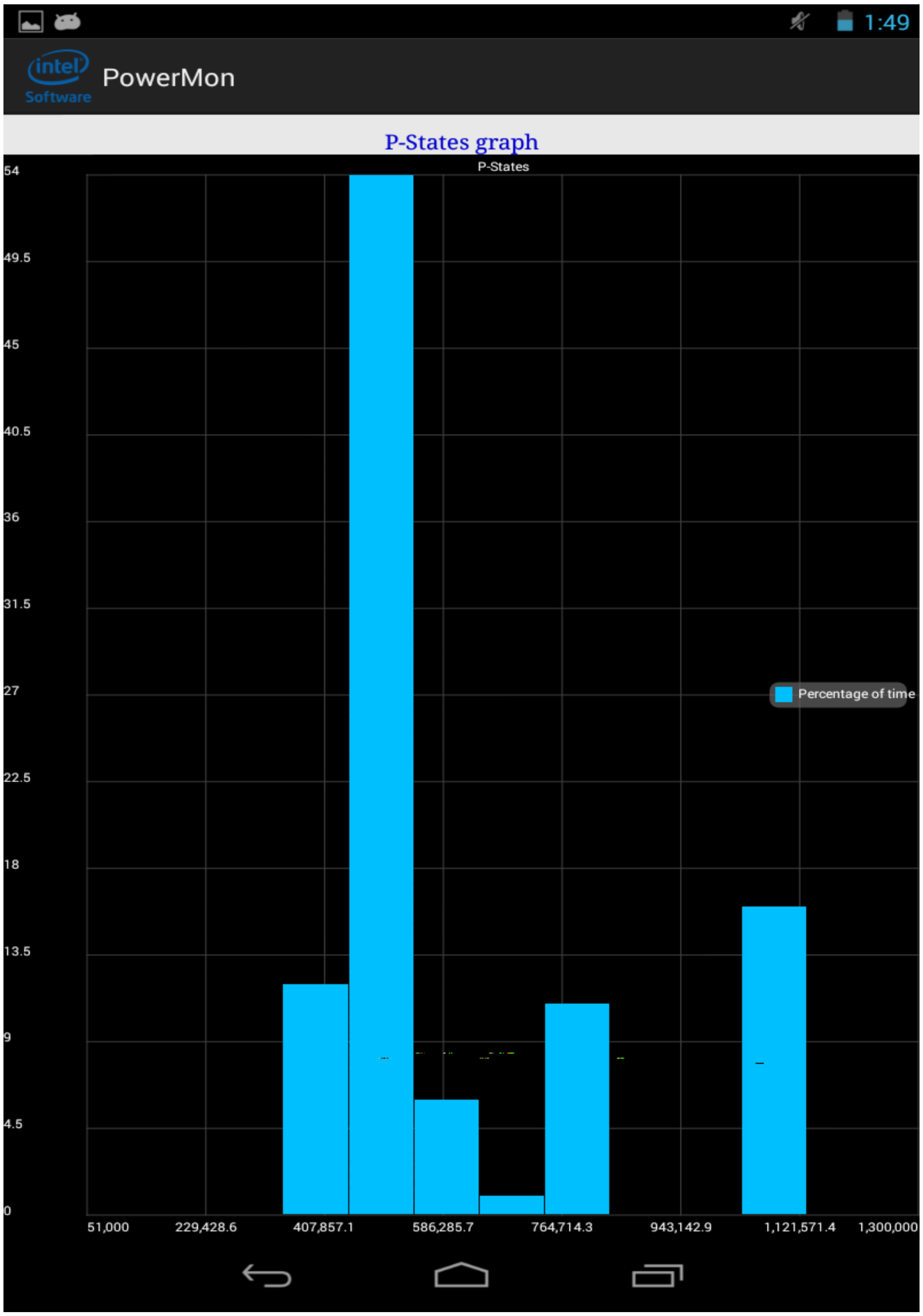
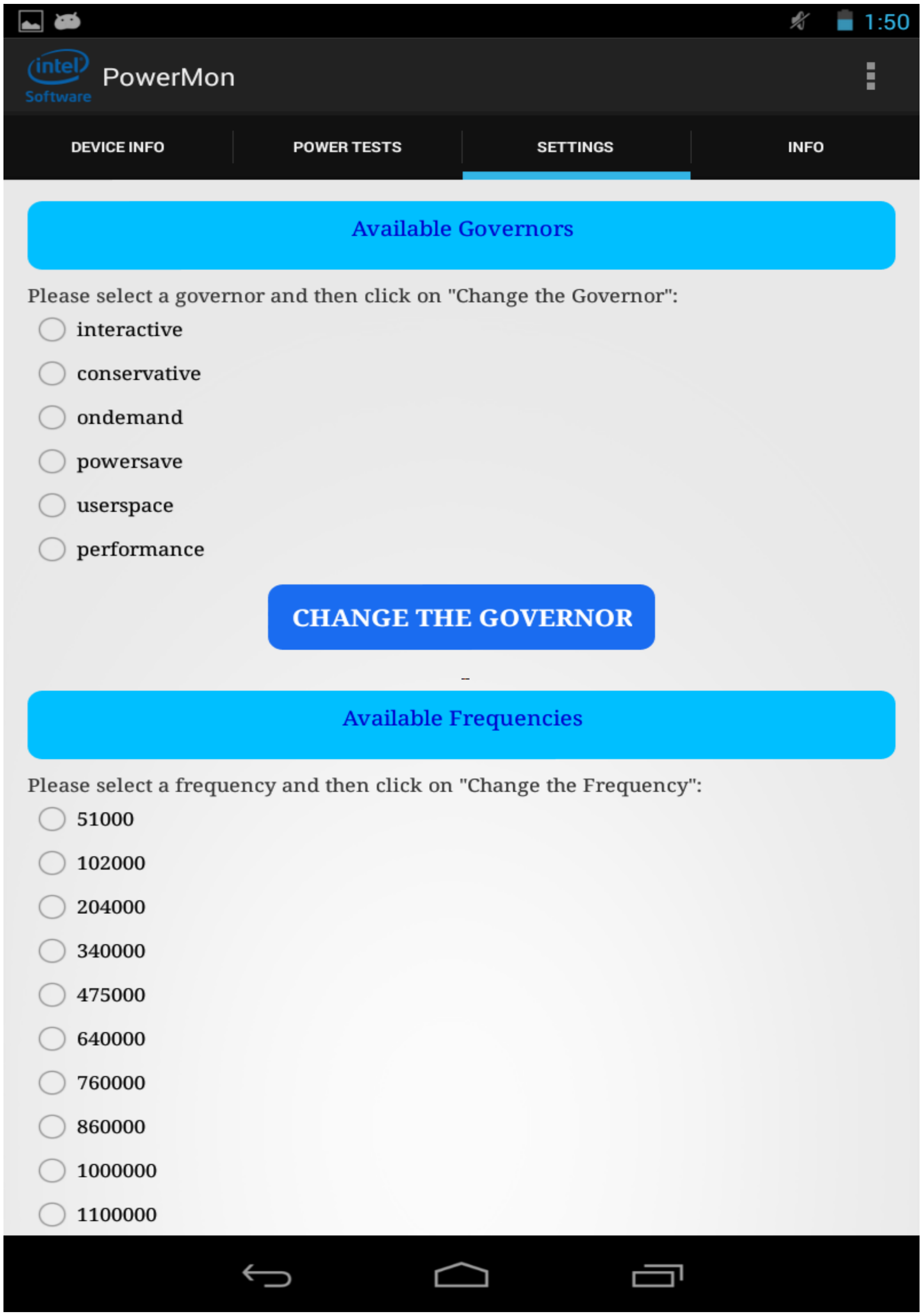


Figure 3.16: P-states results graph.



Please select a governor and then click on "Change the Governor":

- interactive
- conservative
- ondemand
- powersave
- userspace
- performance

Please select a frequency and then click on "Change the Frequency":

- 51000
- 102000
- 204000
- 340000
- 475000
- 640000
- 760000
- 860000
- 1000000
- 1100000

Figure 3.17: Settings screenshot.

tool that can be used by developers and the hardware manufacturer in order to debug the energy efficiency of Android applications and processors with very low overhead. Then, we presented two case studies where we collected and analyzed power data using SoftPowerMon on two different devices for several Android benchmarks. Finally, we compared the performance of SoftPowerMon to two other popular profiling tools and determine that it can collect data with negligible incurred overhead. SoftPowerMon can explain why an application is consuming a specific amount of power as opposed to how much it is consuming.

CHAPTER 4: ENERGY-EFFICIENCY COMPARISON OF MOBILE PLATFORMS: A QUANTITATIVE APPROACH

4.1 Introduction

Mobile devices changed the landscape of computing. Gradually, mobile devices are replacing the traditional personal computers (PCs) to become the devices of choice for entertainment, communication, and productivity. The introduction, or the blooming of these devices, poses a challenge to the industry that provides online tools and services. All of a sudden, for instance, company X, which has a website that traditionally provided a service to stream videos, is now ”forced”—in order to remain competitive in the industry—to provide tailored apps for each mobile operating system (OS). As a result, due to the fact that each OS has its own programming environment, executives must decide whether to provide N amount of apps for N amount of operating systems or perhaps create a single mobile-web-based app that can be accessed via a browser.

Mobile devices are limited by their battery life, which is only growing by 5% annually [73]. Due to the scarcity of battery life, users are not simply evaluating an app from a performance perspective, as they used to for desktop apps, but they are also evaluating the apps from an energy efficiency perspective. As a result, in order for an application to succeed in the market, it needs to be optimized from an energy-efficiency perspective.

4.1.1 Goals

Since mobile apps are evaluated from an energy efficiency perspective, we performed energy efficiency comparison of mobile platforms using a quantitative approach in order to achieve the following:

1. Since developing an app for each OS is costly, our goal is to determine if it is worthwhile from an energy-efficiency perspective to develop native apps or to simply develop web-based apps. As a result, we compare the energy efficiency of native versus web-based apps.

2. Since apps are evaluated from an energy-efficiency perspective, our goal is to determine if same categories of applications show similar impact on the energy consumption of the platform on the top three most popular operating systems.

3. Through detailed evaluation of energy efficiency of apps, our goal is to show how the power metrics correlate in order to describe the causes of the energy inefficiencies of apps.

4. Through observation of energy consumption behavior of apps, our goal is to derive a list of recommendations for app developers.

4.1.2 Contribution

The contributions of this chapter are listed as follows:

- We developed a tool *EnergyMeter*, which can collect the platform, package, core, and GPU energy consumption on Windows platforms.

- We characterize the popular mobile apps into eight categories: browsers, video streaming, music streaming, maps, video chatting, cloud storage, social networking, and e-mail services.
- For each mobile app category, we profile the energy efficiency on the three most popular operating systems, Windows, iOS, and Android, in terms of the following list:
    1. Same app categories on a single operating system (OS) compared to other apps of the same category on the same OS.
    2. Ranking of the energy efficiency of the same app on a different OS.
    3. Comparing the energy efficiency of the same application if accessed using a native app or using a browser.
- Based on our case studies, we derived a list of observations, causes, and implications, which are summarized in Table 4.1.
- We provide a list of the top 10 energy-efficiency rules recommended for app developers.

Table 4.1: List of observations and implications based on the case studies comparing the energy efficiency of applications on three mobile operating systems: Windows 8 and beyond, iOS, and Android.

Observations	**Causes**	**Implications**
The major 3 operating systems providers		
O1: Apps released by Apple are more energy efficient than all third-party apps belonging to the same app category.	The energy profiling tool supplied by Apple contained the least precise information compared to other profiling tools.	Developers need more variety of metrics to be profiled and with higher precision in order to better profile their apps.
O2: Apps released by Google were more concerned with performance than with energy efficiency.	Google apps on Windows 8.1 changed the timer resolution from 15.6 ms to 1 ms, causing high average of wake-ups per second. Google apps on Android acquired the highest number of wakelocks and wifilocks.	Developers for Google apps should attempt to increase the energy efficiency of their apps by enabling the platform to go to idle state (e.g, refrain from changing timer resolution on Windows 8.1 and reduce the numbers of wakelocks and wifilocks on Android).

Continued Table 4.1 on next page

Observations	Causes	Implications
O3: Apps released by Microsoft were more concerned with energy efficiency than with performance	Apps performed poorly but ranked high in energy efficiency. (For example, streaming music kept interrupting the music while buffering)	Developers should be capable of balancing energy efficiency and performance. Apps with high energy efficiency and low performance cannot compete in the mobile apps market.
Observations related to energy-efficiency application design		
O4: Multi threading can lead to either increasing the energy efficiency of an app or decreasing it.	If threads are not concurrently executing a task, then there will be an imbalance in the utilization of cores, leading the processor's package to remain for a high percentage of time in active state.	Multi threading without adequate balance of concurrency can negatively impact the energy efficiency of an app.
O5: Despite the fact that buffering data can enable Wi-Fi radio to go to deep idle sleep states, it can also increase the power consumption of memory.	Large buffer data are stored in memory and caches that consume high power.	Data size of the buffer needs to be carefully examined because developers need to balance between the energy savings from enabling the Wi-Fi radio to go to idle sleep states and the extra energy consumption due to the increase in memory usage.

Continued Table 4.1 on next page

Observations	Causes	Implications
O6: In general, native apps consumed less energy than accessing the web-based version of the app.	Native apps tend to have higher CPU utilization and lower memory utilization compared to web-based apps.	It seems a good investment for companies to create native apps for each platform in order to increase the energy efficiency of their product.
O7: The same app can rank as the most energy efficient in one category, using a specific OS, but can rank as the least efficient on a different OS.	Each OS has its own architecture.	Following the same architectural design with different language implementations in order to provide an app for each platform is not enough. Also, the architectural design should be specific to each platform in order to optimize the energy consumption of the app based on the OS.
Observations related to app developers practices		

Continued Table 4.1 on next page

Observations	Causes	Implications
O8: By comparing applications with the same functionality and running on the same platform, we found that they can vary vastly in terms of energy consumption (more than 50% in some instances).	Due to the lack of point of reference, app developers cannot determine the range of energy efficiency value that they need to target.	There is a need for energy benchmark apps for each category of apps in order for developers to use them as a baseline to compare it with the energy consumption of their apps instead of using device idle energy consumption as the baseline.
O9: Debugging the energy efficiency is a complicated process where one specific energy metric value in a specific context can mean something completely different in a different context.	Example: apps with high wake-up average per second are considered energy inefficient. However, if the processor is active for a large percentage of time and the average number of wake-ups is low, it does not mean that the app is energy efficient.	Developers should not focus on one or two energy profiling metrics to profile the efficiency of their apps (e.g, CPU or memory usage). Adequate profiling requires correlation of extensive set of power metrics and interpreting the data collected in the context of the collection.

Continued Table 4.1 on next page

Observations	Causes	Implications
O10: Changing timer resolution on Windows OS and holding wakelocks on Android OS are a common practice.	Either lack of awareness of the overhead of those energy-inefficient practices on energy consumption or developers are making conscious decision to sacrifice energy efficiency in order to increase performance.	There needs to be more awareness among developers on the impact of these two metrics on the overall energy efficiency of their apps. Energy efficiency of apps should not be an afterthought but it should incorporated in the overall design of the app.

End of Table 4.1

4.1.3 Organization

To this extent, the remainder of the chapter is organized as follows. In Section 4.2, we describe the current three popular mobile operating systems, which are Windows, iOS, and Android. Then, we list and describe in Section 4.3 the top 10 energy-efficient programming rules. We present our quantitative analysis approach in Section 4.4 followed by eight detailed case studies on each platform in Section 4.5. Based on the case studies, we derive a list of implications in Section 4.6. Then, we examine related work in Section 4.7 and conclude in Section 4.8.

4.2 Mobile Device Operating Systems

There are many mobile device platforms. According to an article published by CNET news [21], iOS, Android, and Java ME held the top 3 positions for the mobile and tablet worldwide market share of operating system usage for November 2013 as shown in Figure 4.1. However, according to Gartner forecast [5], the top 3 positions for the mobile devices by operating system worldwide will be Android, iOS, and Microsoft in the next couple of years as shown in Figure 4.2. Therefore, we focus in this chapter on the later operating systems (OS).

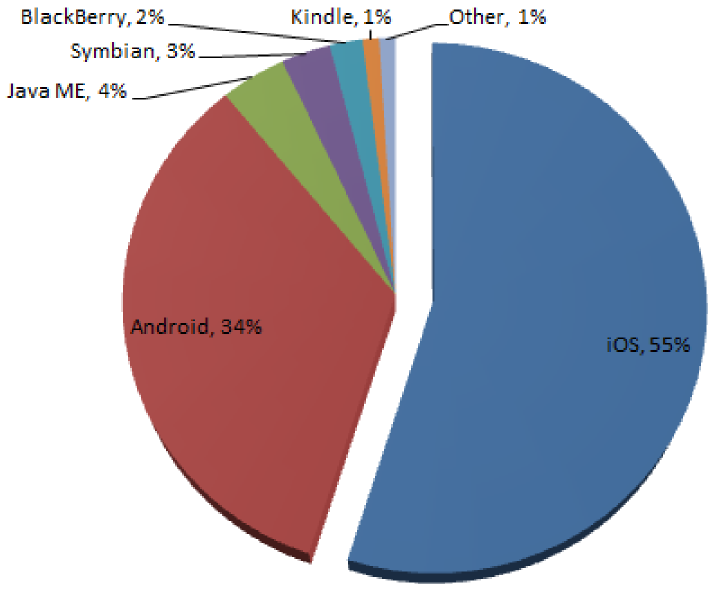


Figure 4.1: Mobile and tablet worldwide market share of operating system usage for November 2013. Net Market Share collects browser data from a worldwide network of over 40,000 websites.

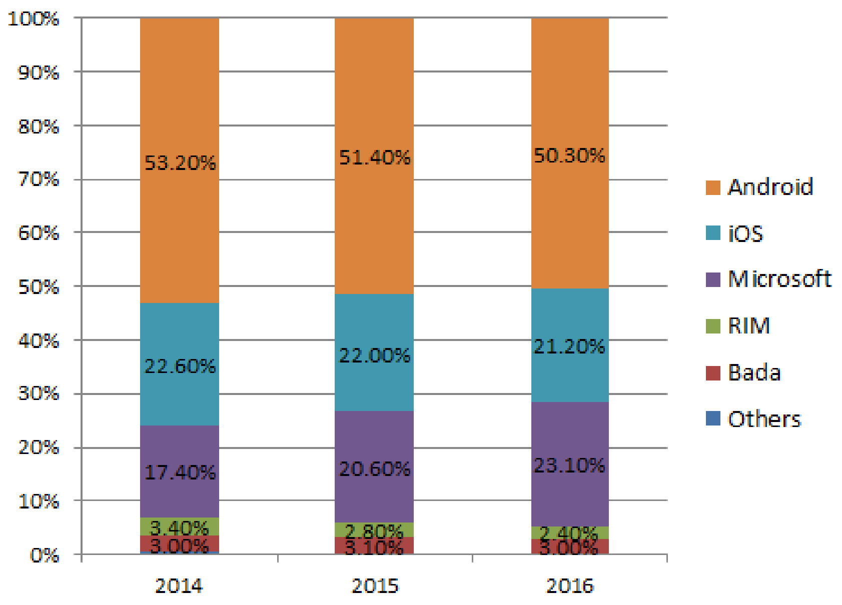


Figure 4.2: Gartner forecast of mobile devices by open operating system, worldwide, 2014-2016.

4.2.1 Windows 8 and Beyond

Windows 8 and beyond (Windows 8.1) was developed by Microsoft. This operating system was developed in order to compete with the tablets market, which is currently dominated by iOS and Android. It is significantly different than all previous OS released by Microsoft. It still has the desktop feature, which is associated with personal computers, in addition to the adoption of the *Metro* design concept where applications are represented as *tiles* that have a similar look and feel as other mobile devices with the exception that they are a bit larger in size.

One of the advantages of the new OS feature is the introduction of *Windows Push Notification Services (WNS).* WNS enables Windows Store Apps developers to sent toast, tile, badge, and raw updates from their own cloud service to the WNS server [18]. The benefits of WNS are as follows:

1. No persistent socket between all apps and the remote server. Instead, Windows maintain the connection to WNS server. As a result, this mechanism reduces

the overhead of sending keep alive messages to the remote servers.

2. Single connection between client and cloud service which can support all apps.

3. No need to maintain many parked TCP socket connections.

4. Apps do not have to reside in memory at all times but yet they keep getting updates from the server.

Another key feature enabled by Microsoft through the release of Windows 8 is the *connected standby* feature, which is the connected standby power model that runs at a very low power level in order to stay connected and up-to-date even when the device appears to be powered down.

The other specific feature of Windows mobile devices is *timer resolution*. The timer resolution determines the time interval for the OS to perform two actions. The first action is to update the timer tick count if a full tick has occurred, and the second one is to check if an already scheduled time has expired. The current default timer resolution is set to 15.6 ms. Some applications may change the timer resolution to a low value, causing the platform to consume more energy over time due to frequent wakeups. For example, if the timer resolution is set to 15.6 ms, then the platform will have 64 calls per second. However, if an application changes it to 1 ms, then the platform will have 1,000 calls per second. Changing the timer resolution is not always a negative practice. If an application is actively using the processors, then changing the timer resolution may not have a bad impact since the processor is already in active state upon the timer duration expiration.

4.2.2 iOS

iOS is the mobile operating system introduced by Apple in 2007. It was derived from OSX (Apple's desktop OS) because its kernel is based on Darwin OS. iOS has the following four architectural layers [8] (from highest to lowest):

- **Cocoa Touch Layer:** This layer contains the key frameworks for building iOS apps. They define the appearance and provide the infrastructure for multi-tasking, touch-based input, push notifications, address book UI, iAd framework, messages UI, and Map Kit framework.
- **Media Layer:** This layer contains the graphics, audio, and video technologies that enable developers to implement multimedia apps.
- **Core Services Layer:** This layer contains the fundamental system services for apps. These services include peer-to-peer services, which enable the initiation of communication sessions between nearby devices. It also includes iCloud storage, which enable apps to write data to a central location on the cloud. In addition, it provides a service for automatic reference counting (ARC), where the compiler manages the lifetimes of objects instead of having the developer worry about that aspect of their apps. Moreover, it provides the data protection service that developers can use in order to encrypt the user sensitive data.
- **Core OS Layer:** This layer contains the low-level features that most other technologies are built upon. These are not used directly by an app, but they are used by the upper layer frameworks.

iOS Software Development Kit (SDK) contains the tool and interfaces needed to develop, install, run, and test native apps. The native apps are developed using iOS SDK and Objective-C language.

4.2.3 Android

Android is an open-source operating system that is based off of a modified Linux Kernel. Android has four architectural layers which are from the highest to the lowest: applications, application framework, Libraries and Android runtime, and Linux kernel [55].

- **Applications:** This is the layer that most users interact with. It is capable of running Java applications.

- **Application Framework:** This layer is written in Java. It provides the structure for all running applications. Some of the functions of this layer are the activity manager and call manager.

- **Libraries:** Libraries are also known as APIs. They are written in either C or C++. They include functionality such as database storage and graphics APIs.

- **Android Runtime:** This layer consists of the custom virtual machine known as Dalvik Virtual Machine and the core libraries which are needed in order to operate the Java code.

- **Linux Kernel:** This layer includes the set of drivers for the hardware components such as display, keypad, and connection for Wi-Fi and cellular signals [93].

In order to develop apps for Android, developers need Eclipse, Java Development Kit (JDK), Android Software Development Kit (SDK), and the Android Development Tools (ADT).

A specific feature for Android is the presence of *wakelocks*. The wakelock is a mechanism that informs the OS that the app needs the device to remain on. There are two classes of wakelocks: kernel space and user space [75]. These wakelocks may decrease the energy efficiency of an app because they keep the device on. Wakelock don't simply keep the device on while the screen is on; if developers forget to release the lock, the app will remain awake even if the power button is pressed.

Another specific feature for Android is the presence of another lock called *WifiLocks*. This type of lock enables the application to keep the Wi-Fi radio awake until the release of the lock. WifiLock is not exclusively acquired by a single app at a time; however, multiple apps can acquire it concurrently and the Wi-Fi radio remains in full power state until all apps release the lock. This type of lock can be very expensive in terms of energy consumption, especially if a developer forgets to release it.

4.3 Top 10 Energy Efficient Programming Rules

There are general rules that software developers should adopt in order to ensure that their apps are energy efficient. The 10 rules are as follows:

1. **Extend platform sleep duration**

   In order to increase the energy efficiency of a platform, developers should keep the platform's components active for the shortest possible duration and avoid waking up components unless necessary. For example, when developing apps for Windows OS, developers should avoid decreasing the timer resolution interval in order to avoid frequent wake-ups of the platform. Another example is using the OS's API, which can extend sleep duration. For instance, Windows provides two APIs, *SetWaitableTimerEx* and *SetCoalescableTime*, which can be used in order to decrease unnecessary wake-ups.

2. **Event-driven architecture**

   Polling can cause unnecessary wake-ups for the platform. As a result, developers should use event driven interrupts instead of polling for information.

3. **Design energy-efficient user interface (UI)**

   Energy-efficient UI consists of accelerating user interaction. In other words, users should be able to get to the required screen using the least amount of clicks. In addition, frequent screen changes can impact the energy consumption of the display and GPU. Therefore, developers should balance the interface experience with the energy efficiency of rendering the updates. Moreover, developers should consider whether to display a progress bar (which keeps getting updated) versus a simple busy indicator (which doesn't require interface update). Finally, when developers are considering graphically rich interfaces, they should weight the impact of their creativity on the energy efficiency of the app. For example, even though splash screens can be graphically rich, they can have high energy consumption.

4. **Consider data locality**

   Memory and storage are high power consuming components of a mobile device. Therefore, developers are encouraged to operate on small data at a time so data can stay in cache, keep the percentage of memory usage low, and keep the number of reads from storage low.

5. **Efficient multithreading**

   Multithreading within apps is a common practice. However, multithreading does not necessarily mean concurrency. But, unless threads are concurrently running on a multi-core device, then they won't be optimized in terms of energy efficiency. Figure 4.3 visualizes the impact of thread execution sequence on the package active duration. Ideally, a developer needs to target to enable the package to enter in idle sleep state for the longest possible duration.

6. **Take advantage of context programming**

   Context programming makes the apps smarter. Developers should enable their apps to sense the environment in which they are operating and trigger reactions based on the changes in the environment. For instance, they should be able to sense if the device is in AC or DC mode. If the device is connected to power, then they can trigger backup.

7. **Be aware of low update frequency intervals of sensors**

   Today's mobile devices are geared with several sensors such as an accelometer and gyrometer. These sensors are managed through APIs that enable developers to change the interval update frequency. Developers should avoid low update intervals of these sensors in order to reduce the number of wake-ups caused by the updates and enable the sensors to go to idle sleep states.

8. **Coalescent network activities**

   Every connection to the network consumes power to transmit or receive packets in addition to the tail power of the network device. As a result, spacing connections out unnecessarily can waste significant power. Developers may reorganize their code in order to group the app's network connection together.

9. **Close network sockets**

   After an app finishes transmitting or receiving data over the network, it does not automatically close the connection. As a result, after an interval with no network activities, the Wi-Fi radio enters an idle sleep state until the server will time out and close the socket by sending a FIN packet, which will switch the Wi-Fi radio back to active state. Therefore, it is highly recommended that developers close network sockets when they are done with transmitting or receiving data.

10. **Avoid high-resolution images**

    High-resolution images are pretty and may have an added benefit to the user experience and feel of an app. However, these high-resolution images are costly, in terms of energy efficiency, to render. Therefore, developers need to weight the importance of high-resolution images versus energy efficiency of their apps.

Device	OS Version	Processor	Memory	Storage
Nexus 7	Android 4.3 Kernel 3.1.0-g9e52a21	Qualcomm Snapdragon S4 Pro APQ8064-1AA x4	1 GB	16 GB
iPad Air	iOS 7.0.6	A7 chip with 64-bit architecture and M7 motion coprocessor	1 GB DDR3	16 GB
Surface 2 Pro	Windows 8.1	Intel(R) Core(TM) i5-4200U (HASWELL ULT)	4 GB	64 GB

Table 4.2: List of devices

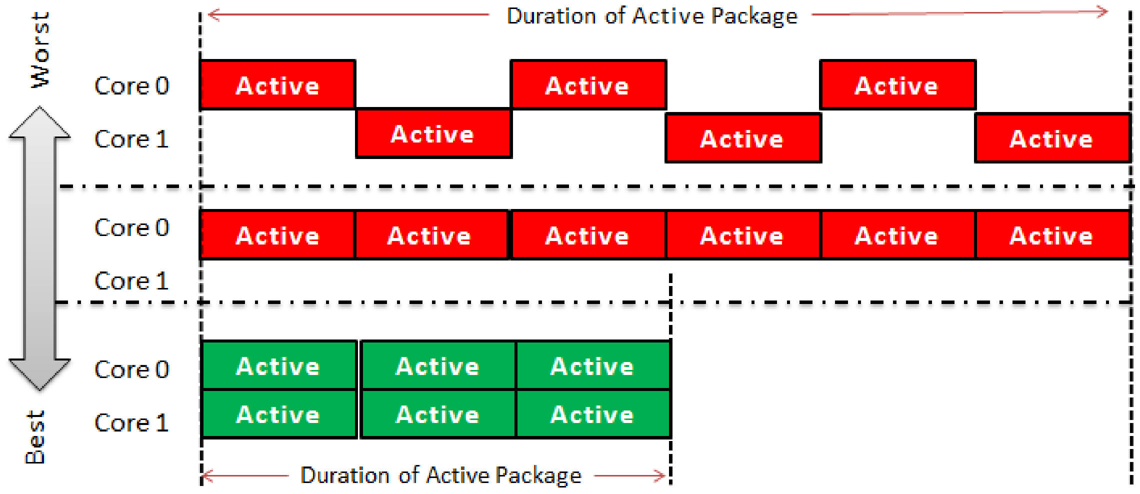


Figure 4.3: Comparison of the impact of multithreading on package active duration.

4.4 Quantitative Analysis Approach

Our experimental setup includes three devices as shown in Table 4.2. For each OS, we used different tools in order to collect the power metrics.

4.4.1 Windows

We installed Windows Assessment and Deployment Kit for Windows 8.1. We used Windows Performance Toolkit tools: Windows Performance Recorder (WPR) [23] and Windows Performance Analyzer (WPA) [22]. In addition, we have used Event

Tracing for Windows (ETW) [11]. During our collection, we examined the following metrics:

1. **Per-core idle sleep states:** This is the percentage of time spent in each idle sleep state per core.

2. **Package idle sleep states:** This is the percentage of time spent in each idle sleep state during the test duration. In order for package to switch to a sleep state, it requires that both cores concurrently switch to a sleep state.

3. **Core frequency:** This is the power state of the core when it was in active state. Since the platform we used had two cores and each had independent frequency value, we first summed the duration spent in each frequency per core, and then calculated the average.

4. **Hit count and total active duration:** Hit count is the total number of times a process was scheduled to run, including all context switches, not just the ones that occurred right after a wakeup. Total active duration is the value of "busy period" where the process was actively processing some "job".

5. **Package and core wake-ups:** We were able to also collect the number of times that package and core transitioned from any idle sleep state to an active state.

6. **Timer resolution percentages:** This metric provides the percentage of time spent in a specific timer resolution interval.

7. **Number of threads:** We were able to determine the number of threads per application.

In order to collect energy consumed by the platform in addition to the energy consumed by package, core, and GPU, we developed *EnergyMeter*, a tool that can collect all the listed metrics.

EnergyMeter Description

EnergyMeter was developed in C++. It takes as an input the test duration and outputs the total energy consumed by the platform, package, cores, and GPU in joules.

In order to collect the platform power, we relied on Windows API, which enabled us to get a handler to the device interface of the battery in order to collect *BATTERY_QUERY_INFORMATION*, which contains all of the battery capacity. The battery capacity *(C)* is reported in milliwatts per hour (mW/h). This capacity value represents the energy stored in the battery. Therefore, *EnergyMeter* collects the capacity at the beginning of the execution of the tool and then after the timer expires. Next, the delta of the two capacity values represents the total energy consumed. Next, we calculate the total energy consumed in joules as shown in Equation 4.1.

$$E_{(j)} = \Delta C_{(mWh)} \times \frac{1}{\underbrace{1000}_{\text{Convert to Watt}}} \times \underbrace{3600}_{\text{Convert to Seconds}} \tag{4.1}$$

In order to collect package, core, and GPU energy consumption, we relied on hardware counters since Surface 2 Pro contains an Intel Haswell ULT chipset that supports energy counters. The processor supports four non-architectural Machine Specific Registers (MSRs) for Running Average Power Limit (RAPL) [15]. The first one is *MSR_RAPL_POWER_UNIT*. This register contains power units from bits (3:0), energy status units from bits (12:8), and time units from bits (19:16). The remaining ones are *MSR_PKG_ENERGY_STATUS*, *MSR_PPO_ENERGY_STATUS*, and *MSR_PP1_ENERGY_STATUS*, which report package, core, and graphics actual energy consumption. The MSRs are updated at approximately 1-ms intervals and the register wraparound time is about 60 seconds when power consumption is high.

In order to be able to read MSRs, the application must run at the kernel level (Ring0). Therefore, upon executing *EnergyMeter*, we initialize the driver and read the power unit MSR determine the energy units. Then, at a 30-second interval, we collect the energy MSRs. In order to calculate the energy used by package, core, and graphics, we calculated the ΔE_{MSR} and multiply it by the energy unit retrieved from *MSR_RAPL_POWER_UNIT*.

Finally, using this tool, we were able to collect the energy metrics for platform, package, core, and GPU.

4.4.2 Android

In order to power profile our Android device, we used the Trepn profiler provided by Qualcomm [20]. Trepn is a diagnostics tool that enables users to profile both performance and power consumption of Android applications which are running on devices with Qualcomm Snapdragon processors. We were able to collect the following metrics:

- **CPU utilization:** Trepn provides the total percentage of CPU utilization overtime per application. As a result, we used that data in order to visualize the changes of load overtime in addition to calculating the average CPU utilization.
- **Average power in uW:** According to Trepn's manual, the average power is calculated by first collecting the power consumption for 5 seconds and using the average value as a baseline denoted as P_{base}. Then, for the remainder of the test duration, the tool collects the power consumption of the device, and then get the average of the power consumption denoted as P_{test}. Finally, the average power is calculated by subtracting P_{base} from P_{test}.
- **Average virtual memory:** The tool also provides per application the size of average virtual memory utilized in MB.
- **Wakelocks:** The tool provides the acquired wakelocks overtime per applications. As a result, we counted the total wakelocks requested per application.
- **Number of threads:** This is the number of threads per app.

- **Wifilocks:** The tool provides the acquired wifilocks overtime per application. As a result, we counted the total wifilocks requested per application.

- **GPU load and frequency:** The tool provided the load and frequency of GPU overtime. Unlike the other metrics which were broken down per application, these values are the utilization by the platform.

Due to the extensive overhead we observed during the collection process using Trepn, we also used SoftPowerMon (described in the previous chapter) to collect the processor's idle sleep states and frequency.

4.4.3 iOS

In order to profile on iOS operating system, we used the Instrument tool provided by Apple [7]. This tool has a specific *Instrument* for energy profiling. Energy profiling can be enabled on the device and upon completion, users can connect the device to an Apple computer where the log can be imported into the Instrument for examination. During our collection we examined the following metrics:

1. **Energy level:** The tool provides on a scale of 0 to 20 the energy consumption of the device overtime. They don't provide an exact value for power drawn but they provide the scale where 0 is the least energy consumed, and 20 is the highest possible energy consumed.

2. **Total CPU activity:** The tool provides the total percentage of CPU utilization overtime.

3. **Graphics utilization percentage:** The tool provides the total percentage of CPU utilization overtime.

4. **Network activities:** The tool also provides the total number of packets sent and received along with the total size in bytes for all packets.

For our analysis, we used all the listed metrics to energy profile the apps running on our iPad Air.

4.5 Case Studies

Using the quantitative analysis approach as described in Section 4.4, we selected eight application categories for our case studies. The scenarios selected are: browsers, video streaming, music streaming, maps, video chatting, cloud storage, social networking, and e-mail scenarios. We chose these eight scenarios because they represent the majority of categories used by mobile device users.

For each scenario, we selected a list of most popular applications and ran the identical applications on all platforms, where applicable. For instance, we ran Facebook on all three platforms; however, even though Amazon Instant Video is a very popular app, it does not have a version for Android. As a result, we only ran it on iPad and Surface 2 Pro. Another example is YouTube, which we did not profile the app version on Surface 2 Pro. This was because all currently available metro apps with some YouTube-name flavor claimed to be capable of running YouTube videos, but they were not the authentic app.

We also profiled the native app in addition to the version that can run through a browser. For all the apps running using a browser, we chose Chrome with a single tab open since the latter has a version specific to each platform.

The list of apps and corresponding version per scenario and per platform are listed in Table 4.3. Please note that when we rank the energy efficiency of an application and compare it to the energy efficiency of another application, we always rank them from the most energy efficient to the least energy efficient.

Scenario	Platform	App	Version
Browsers	Surface 2 Pro	Chrome	33.0.1750.146
		Internet Explorer 11	11.0.9600.16518
		Mozilla Firefox	24.0
	iPad Air	Chrome	32.1700.20
		Bing	2.0.2
		Safari	7.0.6
	Nexus 7	Chrome	33.0.1750.136
		Bing	4.2.3.20140303
		Mozilla Firefox	27.0
Video Streaming	Surface 2 Pro	Amazon Ubox Video	2.2.0.153
		Amazon (browser)	Accessed: Feb 8, 14
		Netflix	2.3.0.12
		Netflix (browser)	Accessed: Feb 8, 14
		YouTube (browser)	Accessed: Feb 8, 14
	iPad Air	Amazon Instant Video	2.4
		Netflix	5.1.2
		YouTube	2.2.0
		YouTube (browser)	Accessed: Feb 8, 14
	Nexus 7	YouTube	5.3.32
		YouTube (browser)	Accessed: Feb 8, 14
		Netflix	3.2.1 build 1346
Music Streaming	Surface 2 Pro	Pandora (browser)	Accessed: Feb 9, 14
		Spotify	0.9.7.16.g4b197456
		XBOX Music	2.2.444.0
	iPad Air	Spotify	0.9.3
		iTune	7.0.6
		Pandora	5.2
	Nexus 7	Spotify	0.7.6.357
		Pandora	5.2
		Xbox Music	2.0.40226
Map	iPad Air	Apple Maps	7.0.6
		Google Maps	2.7.4
		Waze	3.7.8
	Nexus 7	Waze	3.7.7.0
		Google Maps	7.0.1
Video Chatting	Surface 2 Pro	Hangouts	1.0.0.2
		Skype	2.4.0.1007
	iPad Air	Skype	4.17.126
		Hangouts	1.3.2
	Nexus 7	Skype	4.6.0.42007
		Hangouts	1.0.2.717155
Cloud Storage	Surface 2 Pro	Dropbox	2.0.0.0
		Dropbox (browser)	Accessed: Feb 9, 14
		Google Drive (browser)	Accessed: Feb 9, 14
		SkyDrive	6.3.9600.16384
	iPad Air	SkyDrive	4.0.1
		Dropbox	3.0.3
		Dropbox (browser)	Accessed: Feb 9, 14
		Google Drive	2.2.3
		Google Drive (browser)	Accessed: Feb 9, 14
	Nexus 7	Google Drive	1.2.563.31
		SkyDrive	1.1
		Dropbox	2.3.12.10
		Dropbox (browser)	Accessed on: Feb 9, 14
Social Networking	Surface 2 Pro	LinkedIn HD	1.0.0.0
		LinkedIn (browser)	Accessed: Feb 15, 14
		Facebook	1.2.0.12
		Facebook (browser)	Accessed: Feb 15, 14
	iPad Air	LinkedIn	86
		LinkedIn (browser)	Accessed: Feb 15, 14
		Facebook	7.0
		Facebook (browser)	Accessed: Feb 15, 14
	Nexus 7	Facebook	6.0.0.28.28
		Facebook (browser)	Accessed: Feb 15, 14
		LinkedIn	3.3.1
		LinkedIn (browser)	Accessed: Feb 15, 14
E-mail	Surface 2 Pro	GMail Touch	1.0.0.46
		GMail (browser)	Accessed: Feb 16, 14
		Windows Mail	17.5.9600.20315
	iPad Air	Apple Mail	7.0.6
		GMail	2.71828.0
		GMail (browser)	Accessed: Feb 16, 14
	Nexus 7	GMail	4.7.2 (967015)
		GMail (browser)	Accessed: Feb 16, 14
		Outlook.com	7.8.2.12.49.2176

Table 4.3: List of apps and corresponding version per scenario

4.5.1 Browsers Scenario

Our first set of case studies corresponds to browsers scenarios, where upon starting the profiling tools on each platform, we started a 3-minute timer, launched the browser (set to default webpage set upon installation time), and kept the screen on until the timer expired. Upon the timer expiration, we stopped profiling and saved the results.

Surface 2 Pro Browsers Results

We profiled the energy efficiency of the following browsers: Chrome, Chrome 3-tabs, Internet Explorer (IE), IE 3-tabs, IE metro, Firefox, and Firefox-3 tabs. Figure 4.4 displays the energy consumption per browser for platform, package, core, and GPU. Based on the results, we can rank their energy efficiency as follows: Chrome, Chrome-3 tabs, IE, Firefox, IE-3 tabs, Firefox-3 tabs, IE metro.

In order to examine the cause of the difference in energy consumption, we examined core idle sleep states, package sleep states, core frequency, total hit count and active duration, average wake-ups per second for package and cores, and timer resolution as shown in Figures 4.5, 4.6, 4.7, 4.8, 4.9, and 4.10, respectively. It is evident that Chrome, Chrome-3 tabs, IE, IE-3 tabs, and Firefox remained in approximately the same percentage of the core and package idle sleep states, whereas, IE metro and Firefox-3 tabs were in relatively much larger percentages.

IE versus IE 3-tabs: Even though IE and IE 3-tabs had similar active C-States, the former one remained for 81.55% and 8.91% in 800 and 2,300 MHz, respectively, whereas the latter remained for 39.33% and 54.95% in 800 and 2,300 MHz respectively. This explains why IE consumed 28.43% less package energy than IE-3 tabs.

Firefox versus Firefox 3-tabs: The other noteworthy observation is that Firefox 3-tabs has less active duration compared to Firefox with a single tab, however, Firefox 3-tabs consumed more energy. We can attribute the difference by examining the number of core and package wakeups, in addition to the timer resolution of both test cases. Even though, Firefox has less active duration, it causes less average wakeups per second to package and core, which enabled the relatively small percentage of active time of the core and package. On the other hand, Firefox 3-tabs spent 50.86% of the time in 1 ms resolution, resulting in a larger average of core and package

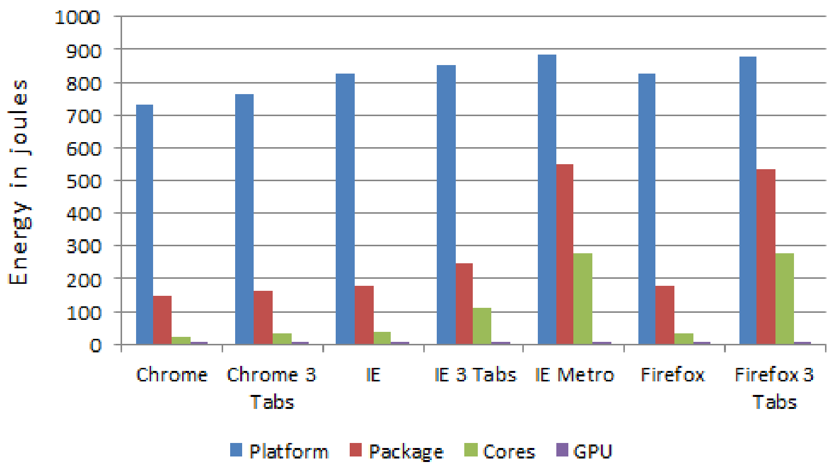


Figure 4.4: Energy consumed by Surface 2 Pro during browsers scenario.

wakeups per second, which in turn increased the package and core active states.

Firefox 3-tabs versus Chrome: The observation of Firefox-3 tabs contradicts, at first glance, with Chrome results which spent 99.81% in 1 ms timer resolution and caused the highest percentage of wakeups while having the same active duration. However, Chrome (1 and 3 tabs) still had a lower percentage of active cores and package compared to Firefox 3 tabs and was much more energy efficient. This contradicting observation was justified once we examined the number of threads for each browser. Chrome had distributed its activities to seven threads, whereas Firefox only had one thread. As a result, Chrome took advantage of concurrency and thus performed the work concurrently on the cores, which enabled both cores to go to sleep for longer duration and thus enabled the package to remain in sleep states for a long duration. On the other hand, Firefox 3-tabs did not take advantage of concurrency, resulting in unbalanced core active duration, leading to a high percentage of package active time, which caused the increase in frequency, which was directly translated to higher energy consumption.

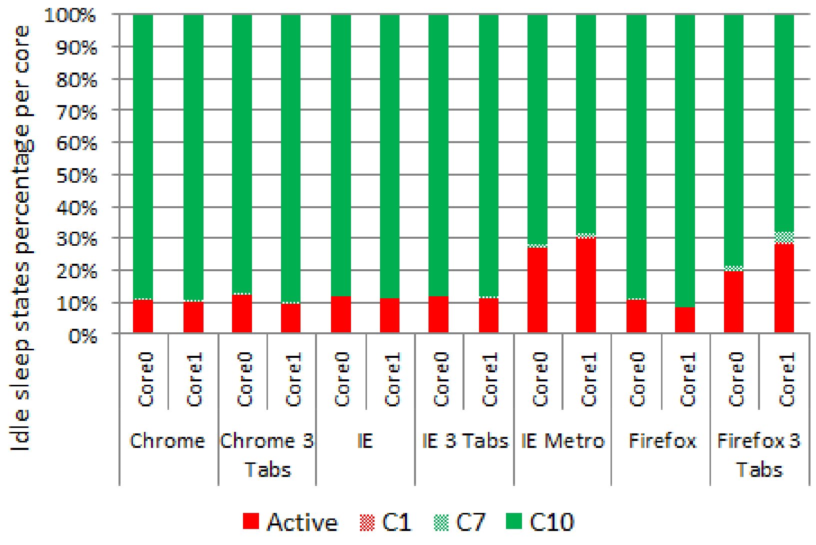


Figure 4.5: Idle sleep states percentage per core collected on Surface 2 Pro during browsers scenario.

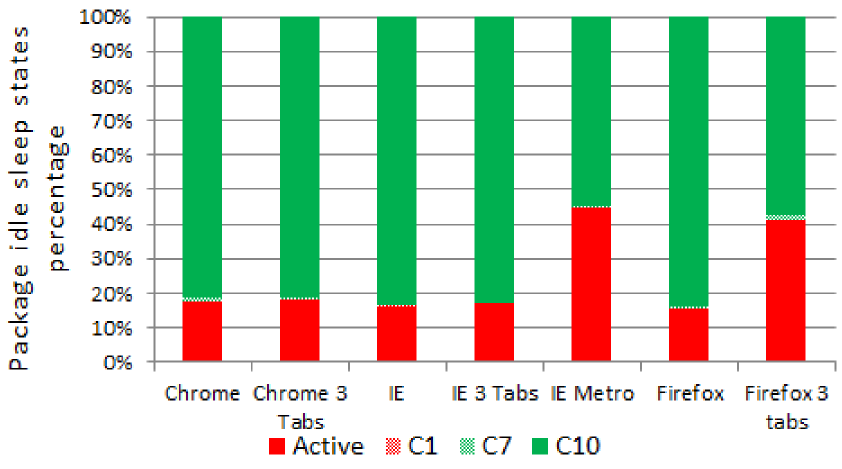


Figure 4.6: Package idle sleep states percentage collected on Surface 2 Pro during browsers scenario.

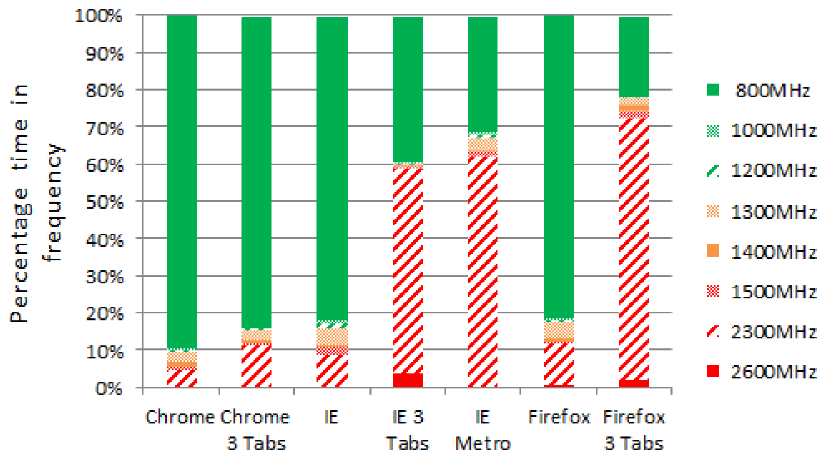


Figure 4.7: Core frequency distribution collected on Surface 2 Pro during browsers scenario.

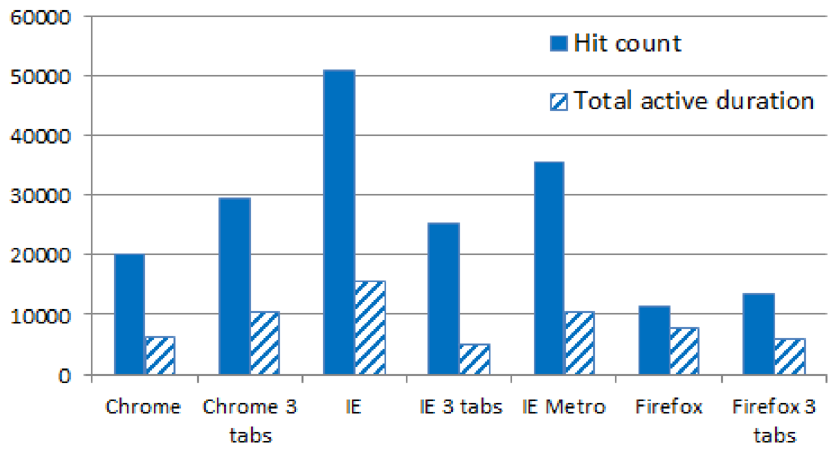


Figure 4.8: Total hit count and busy duration in milliseconds collected on Surface 2 Pro during browsers scenario.

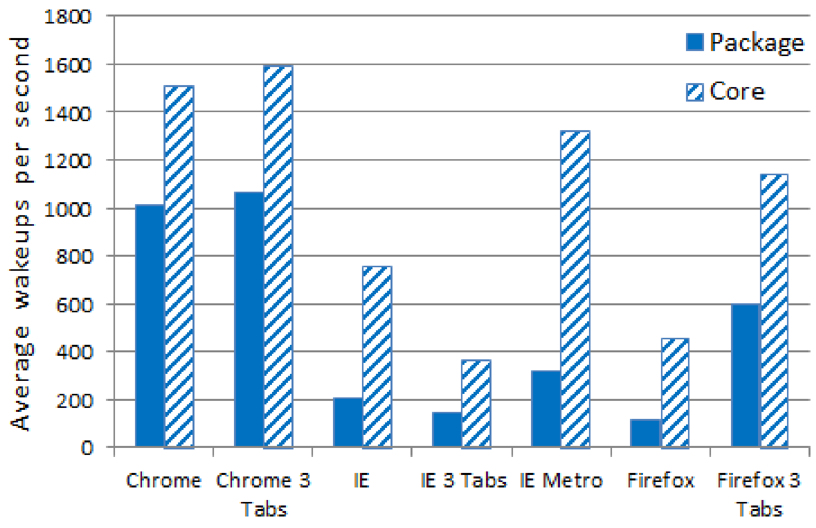


Figure 4.9: Average package and core wakeups per second collected on Surface 2 Pro during browsers scenario.

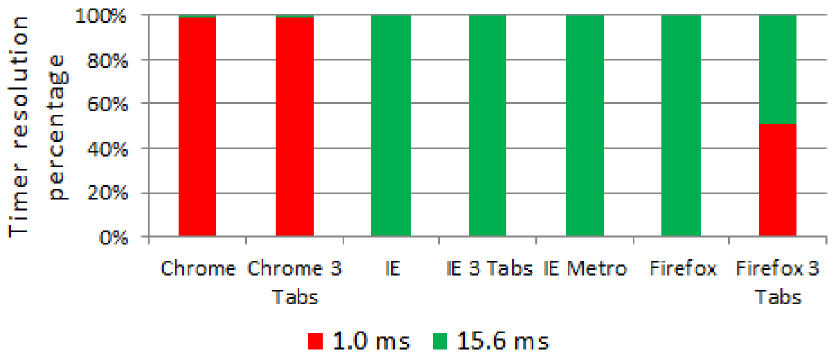


Figure 4.10: Percentage of time spent in each timer resolution interval.

iPad Air Browsers Results

We profiled the energy efficiency of the following browsers: Chrome, Chrome-3 tabs, Bing, Safari, and Safari-3 tabs. Figure 4.11 displays the variation of energy levels during the entire test duration. The average energy level is 3.87, 6.27, 2.65, 1.22, and 1.29 for Chrome, Chrome 3-tabs, Bing, Safari, and Safari 3-tabs, respectively. As a result, we can rank the energy efficiency as follows: Safari, Bing, Chrome. One noteworthy observation is that Chrome consumed 62.05% more energy when we added two extra tabs, whereas Safari only consumed 5.4% more energy when we added the two extra tabs.

In order to examine the cause of the differences in energy consumption, we first examined the total CPU activity percentages and graphics activity percentages for all browsers cases, as shown in Figures 4.12 and 4.13. The average total CPU activities are 3.16%, 3.38%, 7.55%, 2.94%, and 3.29% for Chrome, Chrome 3-tabs, Bing, Safari, and Safari 3-tabs, respectively. The average graphics activities for the same order of browsers are 2.2%, 2.39%, 7.01%, 2.13%, and 2.32%, respectively.

Bing versus Chrome: By examining the results, we noticed that Bing actually had the highest CPU and graphics activities, which at first glance contradicts our first finding that it is the second most efficient browser. As a result, we examine the network activity as well. We noticed that the pattern of network activities is different among browsers. In particular, Chrome received and sent very consecutive large network packets after launching the browser 11,790.31, 4.91, and 12.18 kilobytes in and 885.49, 3.23, and 3.56 kilobytes out. Then, it periodically sent and received very small packets (80 bytes) at an approximate 30-second intervals. On the other hand, Bing sent and received relatively smaller packets after launching the browser. It received in consecutive order 260.9, 194.7, and 90.19 kilobytes and received 11.79, 16.38, and 11.02 kilobytes. Then, it periodically received very small packets (60 bytes) at an approximate 2-second interval.

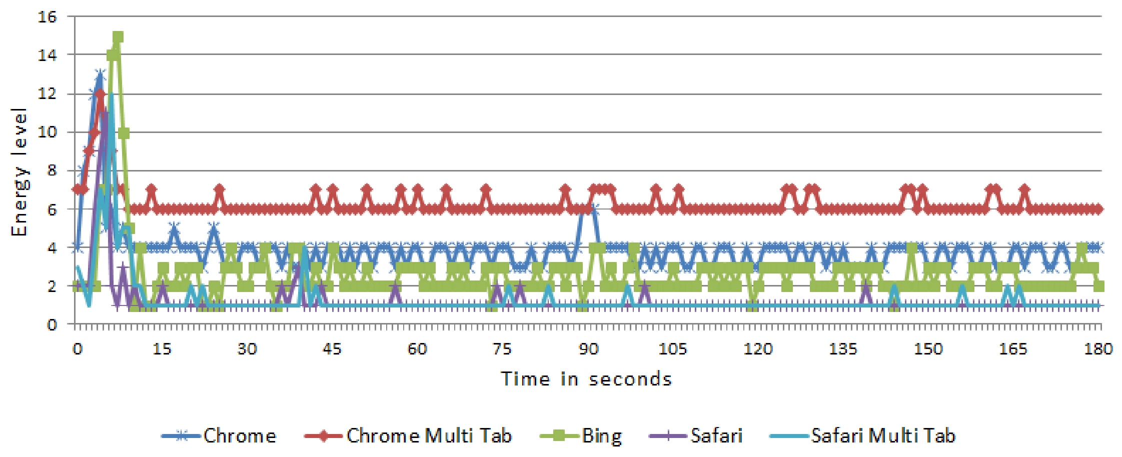


Figure 4.11: Energy level collected on iPad Air 2 during browsers scenario using Instrument.

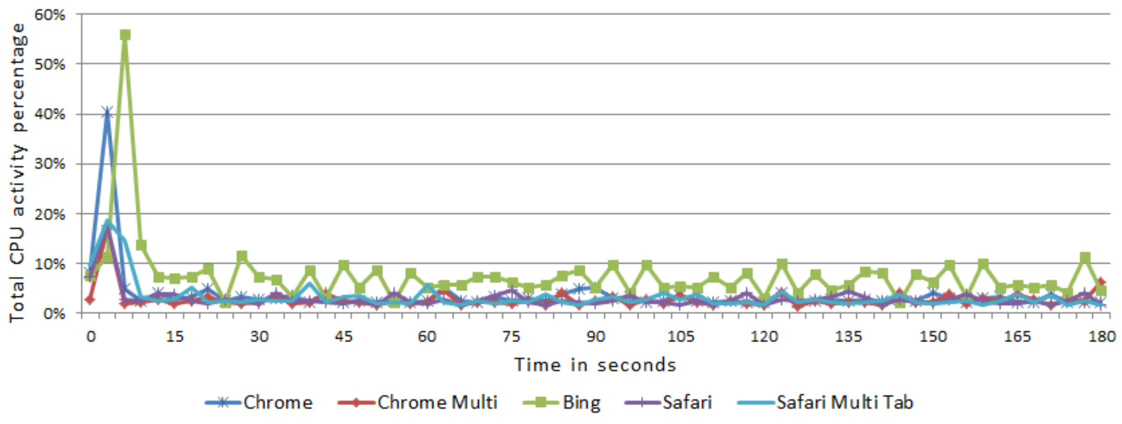


Figure 4.12: Total CPU activity percentage collected on iPad Air 2 during browsers scenario using Instrument.

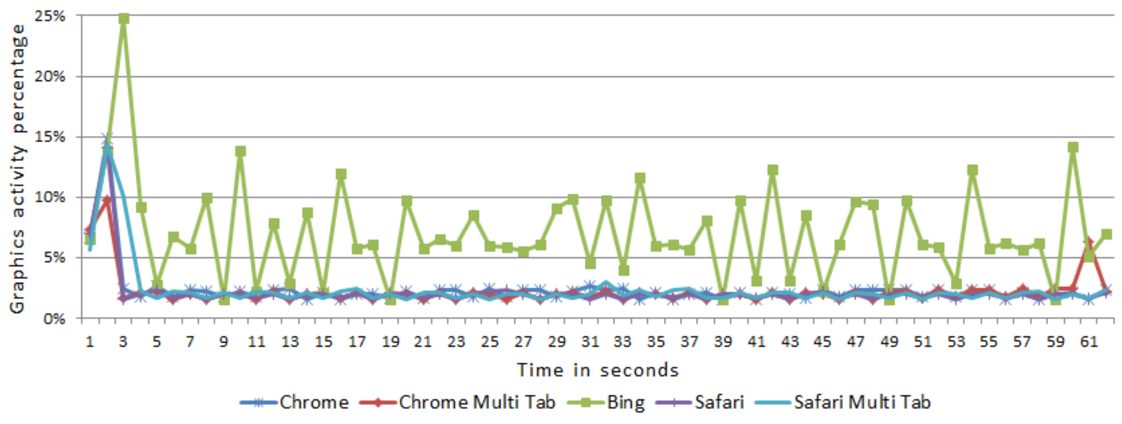


Figure 4.13: Graphics activity percentage collected on iPad Air 2 during browsers scenario using Instrument.

Nexus 7 Browsers Results

We profiled the energy efficiency of the following browsers: Chrome, Chrome-3 tabs, Bing, Firefox, and Firefox-3 tabs. Figure 4.16 represents the percentage of CPU utilization over time, and Table 4.4 represents the power metrics collected using Trepn. Based on the results, we can rank the energy efficiency of browsers as follows: Firefox 3-tabs, Firefox, Chrome, Chrome 3-tabs, and Bing. In order to explain the results, we also examined the core C-states percentages in addition to the core frequency as shown in Figures 4.15 and 4.14, respectively.

Firefox 3-tabs versus Firefox: We observed that Firefox 3-tabs is more energy efficient than Firefox with a single tab because the average CPU utilization increased in the case of 3 tabs, leading to an increase in CPU frequency which lead to an increase in performance, which was translated to less core active duration. The other major observation is the increase of average power consumed between Chrome and Chrome-3 tabs, which was 57.99%. This is due to almost tripling the average CPU utilization, which caused the percentage of high frequency to increase, in addition to the doubling of CPU active duration.

Chrome versus Bing: We also observed huge differences between the average power consumption between Chrome and Bing, where Bing consumed more than triple the amount of power than Chrome. The huge differences can be attributed to the fact that Chome has a higher multithreading index than Bing, which explains why Chrome had a large average virtual memory utilization but very low average CPU utilization. On the other hand, Bing had the lowest amount of threads with the highest CPU average utilization and the lowest virtual memory utilization.

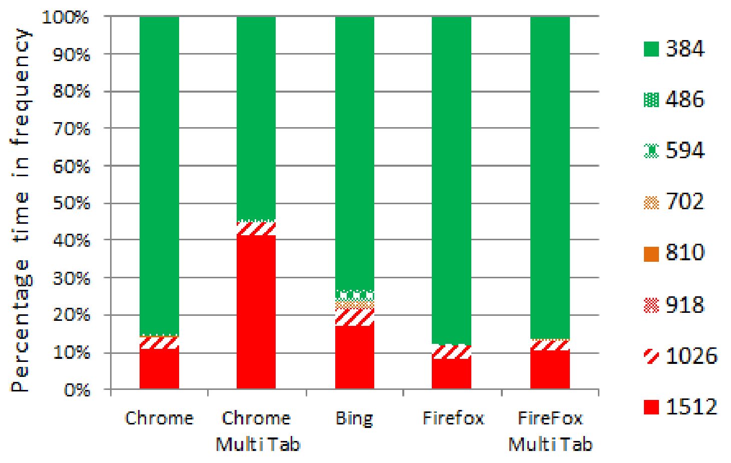


Figure 4.14: Percentage of time spent in each frequency collected on Nexus 7 during browsers scenario using PowerMon.

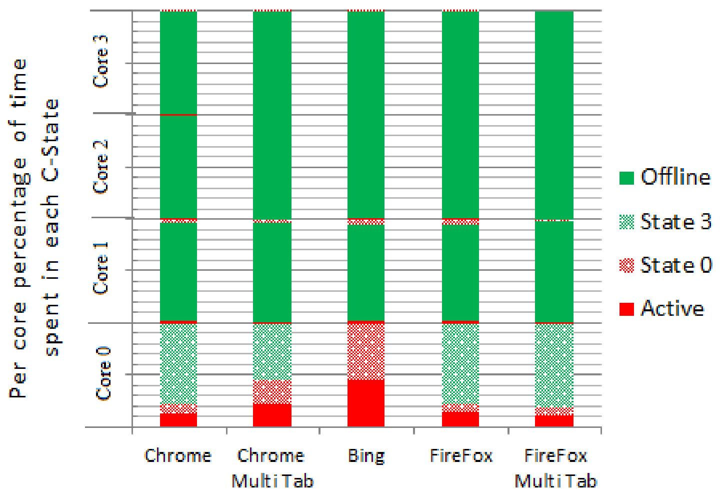


Figure 4.15: Percentage of time spent in each C-State per core collected on Nexus 7 during browsers scenario using PowerMon.

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
Chrome	237,143	0.68	2966.63	88	1171
Chrome 3 Tabs	374,667	1.99	1989	66	1401
Bing	745,017	7.04	912	20	0
Firefox	235,691	0.09	1943	53	0
Firefox 3 Tabs	221,042	0.31	1955	52	0

Table 4.4: Power metrics collected on Nexus 7 during browser scenario using Trepn.

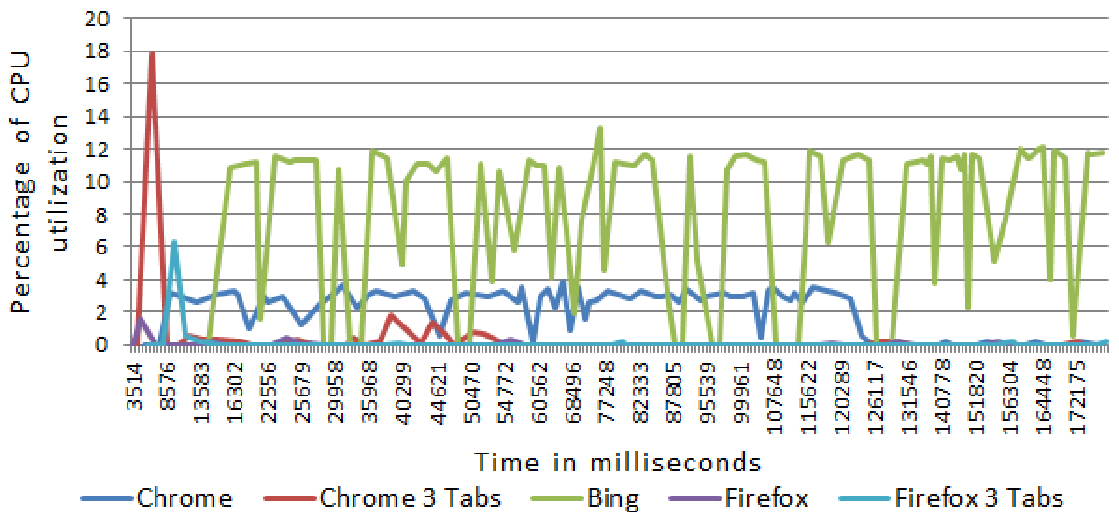


Figure 4.16: Percentage of CPU utilization collected on Nexus 7 during browser scenario using Trepn.

Cross-Platform Comparison for Browsers Scenario

In order to compare the energy efficiency of browser cross platforms, we provide Table 4.5. Even though Chrome was the most energy-efficient browser on Surface 2 Pro, Safari was the most energy-efficient browser on iPad Air, and Firefox was the most energy-efficient browser on Nexus 7.

Application	Surface 2 Pro	iPad Air	Nexus 7
Chrome	Rank 1	Rank 4	Rank 3
Chrome 3-tabs	Rank 2	Rank 5	Rank 4
IE	Rank 3	N/A	N/A
IE 3-tabs	Rank 5	N/A	N/A
IE Metro	Rank 7	N/A	N/A
Firefox	Rank 4	N/A	Rank 2
Firefox 3-tabs	Rank 6	N/A	Rank 1
Bing	N/A	Rank 3	Rank 5
Safari	N/A	Rank 1	N/A
Safari 3-tabs	N/A	Rank 2	N/A

Table 4.5: Cross-platform browsers energy-efficiency ranking.

4.5.2 Video Streaming Scenario

Our second set of case studies are the video streaming scenarios, where upon starting the profiling tools on each platform, we started a 5-minute timer. We launched the app and selected a video, then ran the video in full screen mode until the timer expired. Next, we stopped the collection and saved the results. In the case of accessing the streaming video via a browser, we first launched the browser and typed the user name and password. Then, we started the profiling tool along with a 5-minute timer. Next, we launched the browser and clicked on sign in. We selected the video to stream, clicked on it, and put the video in full-screen mode until the timer expired. Next, we stopped the collection and saved the results. We ran ”The Lorax” using Amazon and Netflix, and we ran ”Elephant Dreams” using YouTube.

Surface 2 Pro Video Streaming Results

We profiled the energy efficiency of the following video streaming apps: Amazon running on desktop, Amazon running in a browser, Netflix running as a metro app, Netflix running in a browser, and YouTube running in a browser. Based on the energy consumption values as shown in Figure 4.17, we can rank the energy efficiency of the video streaming as follows: Netflix metro, Amazon desktop, YouTube browser, Netflix browser, and Amazon browser. In order to explain the differences in energy consumption, we examined core idle sleep states, package sleep states, core frequency, total hit count and active duration, average wake-ups per second for package and cores, and timer resolution percentage, as shown in Figures 4.18, 4.19, 4.20, 4.21, 4.22, and 4.23, respectively.

Amazon desktop versus Amazon browser and Netflix metro versus Netflix browser: By interpreting the results of video streaming test cases, we first noticed that streaming a video using an app consumes significantly less energy when compared to streaming using a browser, as shown in Figure 4.17. For instance, when streaming the same video using the Amazon desktop app, the package and GPU consumed 33.91% and 17.64% less energy, respectively, compared to streaming using a browser. Similarly, when streaming the same video using the Netflix metro app, the package and GPU consumed 39.46% and 11.57% less energy, respectively, compared to streaming using a browser. In addition, since both Amazon and Neflix were streaming the same video "The Lorax," we determined that Netflix is more energy efficient than Amazon.

Amazon versus Netflix: We noticed that even though cores were active for approximately the same percentage of time when comparing Amazon and Netflix, package active time percentage for Netflix is less than Amazon, which means utilization of concurrent cores in Netflix is better than Amazon. In addition, running apps in native mode (desktop or metro), as opposed to browser mode, allowed the cores to be in low frequency value for greater duration than the case when streaming was done via a browser. Moreover, total time in active duration during Amazon and Netflix was only different by 1.04% and 1.18% when comparing both apps during native and browser case,s respectively; however, the average wake-ups per second was much higher in the case of Amazon when compared to Netflix, which explains why Netflix is more energy efficient than Amazon. Finally, the cause of low wake-ups per seconds in the case of Netflix is due to the fact that Netflix kept the timer resolution interval at 15.6 ms, unlike the different apps, which changed it to 1.0 ms.

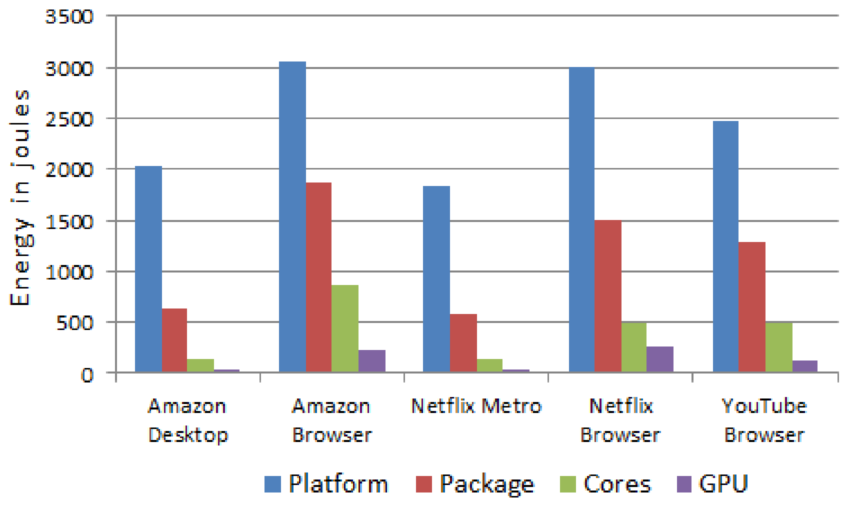


Figure 4.17: Energy consumed by Surface 2 Pro during video streaming scenario.

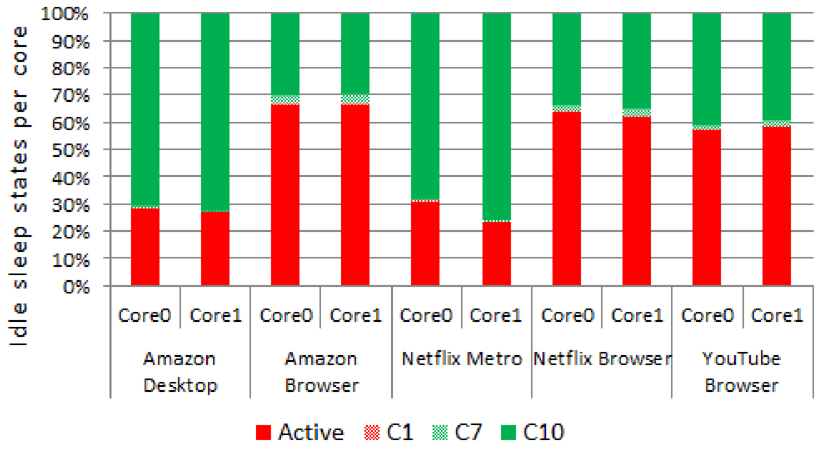


Figure 4.18: Idle sleep states percentage per core collected on Surface 2 Pro during video streaming scenario.

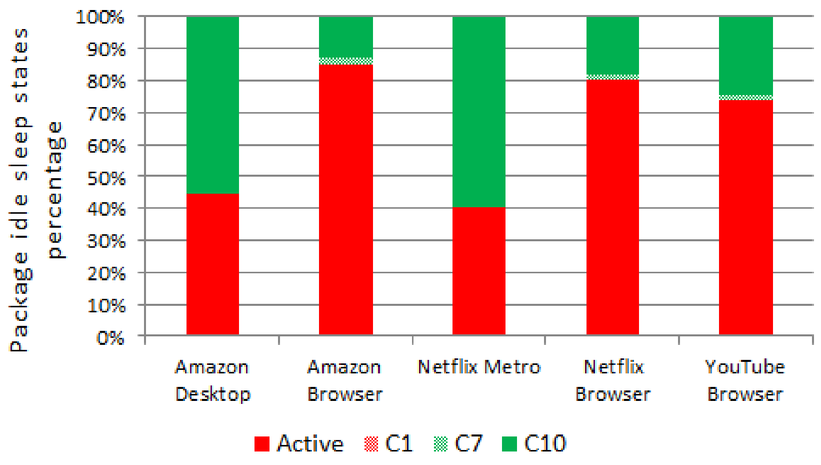


Figure 4.19: Package idle sleep states percentage collected on Surface 2 Pro during video streaming scenario.

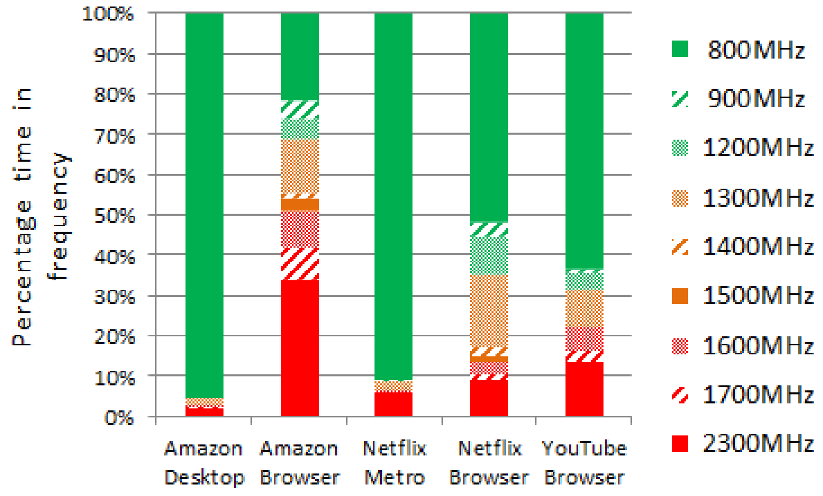


Figure 4.20: Core frequency distribution collected on Surface 2 Pro during video streaming scenario.

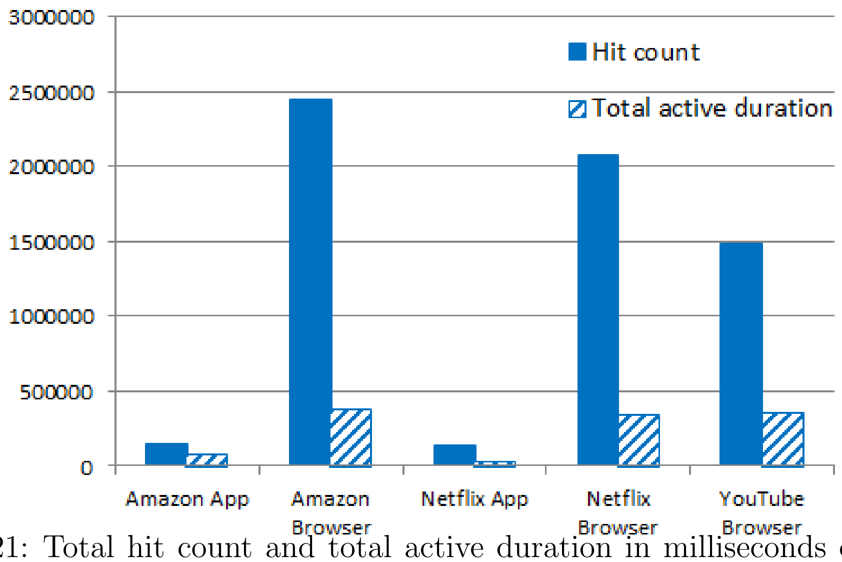


Figure 4.21: Total hit count and total active duration in milliseconds collected on Surface 2 Pro during video streaming scenario.

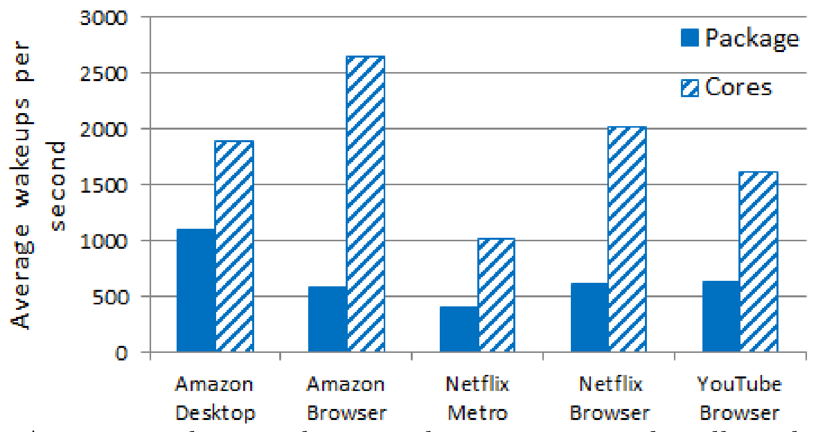


Figure 4.22: Average package and core wakeups per seconds collected on Surface 2 Pro during video streaming scenario.

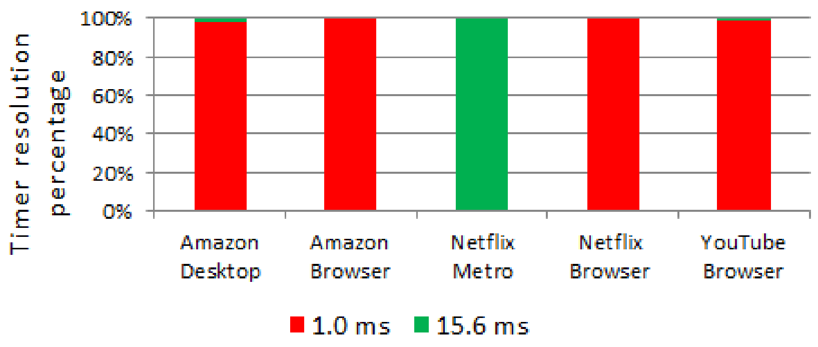


Figure 4.23: Percentage of time spent in each timer resolution interval.

iPad Air Video Streaming Results

We profiled the energy efficiency of the following video streaming apps: Amazon instant movies, Netflix, YouTube app, and YouTube running using a browser. Figure 4.24 displays the variation of energy consumed during the entire test duration. The average energy level is 10.73%, 10.38%, 10.47%, and 10.71% for Amazon, Netflix, YouTube app, and YouTube running using a browser, respectively. As a result, we can rank the energy efficiency of video streaming as follows: Netflix, YouTube app, YouTube browser, and Amazon. By examining the results, it is also evident that streaming a video using a native tool is more energy efficient then running using a browser. In order to examine the cause of the differences in energy consumption, we examined the total CPU activity percentages and graphics active percentages for all our test cases, as shown in Figures 4.25 and 4.26, respectively. The average total CPU activities are 41.56%, 10.23%, 26.16%, and 17.33% for Amazon, Netflix, YouTube app, and YouTube browser, respectively. The average graphics activities for the same order of video streaming apps are: 5.99%, 4.99%, 8.59%, and 7.00%, respectively.

Amazon versus Netflix: By examining the results, we noticed that even though Amazon and Netflix were streaming the same video, the Amazon percentage of CPU and graphics utilization was much higher than that for Netflix. As a result, the difference in CPU and GPU activities percentages can explain why Netflix is more energy efficient than Amazon.

YouTube app versus YouTube browser: Even though streaming the same video using YouTube app was more energy efficient than streaming using a browser, however, streaming a video using a Youtube browser had less percentage of CPU and graphics utilization than streaming via the YouTube app. As a result, we also examined the network activities. We noticed that YouTube app was constantly receiving packets with occasional 0 packets received. On the other hand, streaming using a browser, led to much larger size of packets received at the beginning of the run (due to large buffering of the video), then throughout the test, there were long duration of 0 packet transmissions (20 seconds) followed by a 5 seconds of active receiving. In theory, this method should enable the Wi-Fi radio to go to low-power states for an extended duration, thus reducing the energy consumption of the platform. However, by buffering a large size of data, that led to more utilization of memory, which nullified the savings from the sleep states of Wi-Fi radio and instead lead to causing more energy consumption of the platform.

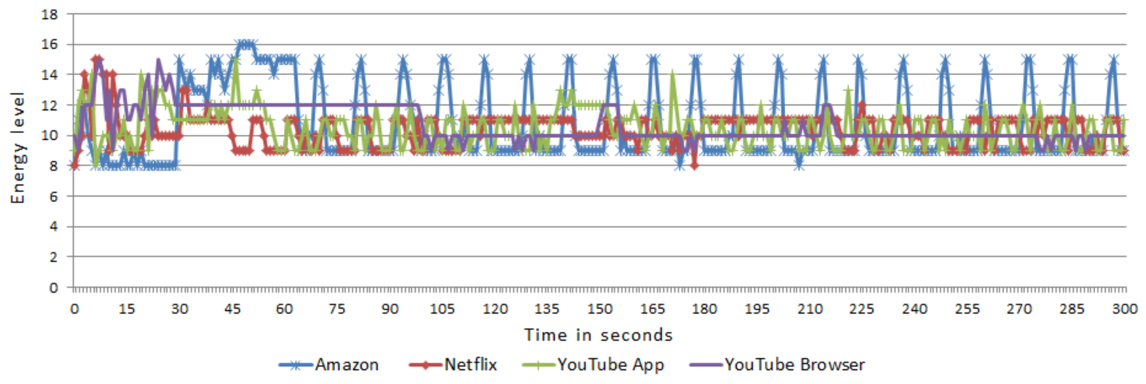


Figure 4.24: Energy level collected on iPad Air 2 during video streaming scenario using Instrument.

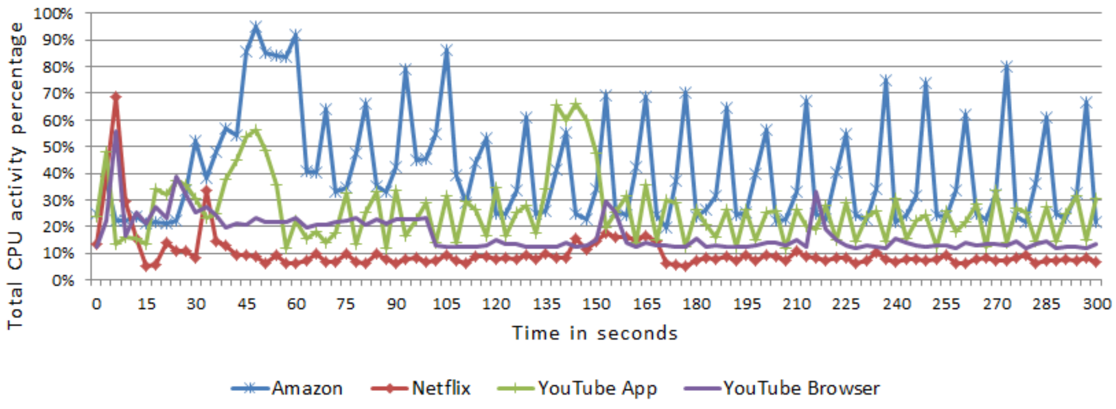


Figure 4.25: Total CPU activity percentage collected on iPad Air 2 during video streaming scenario using Instrument.

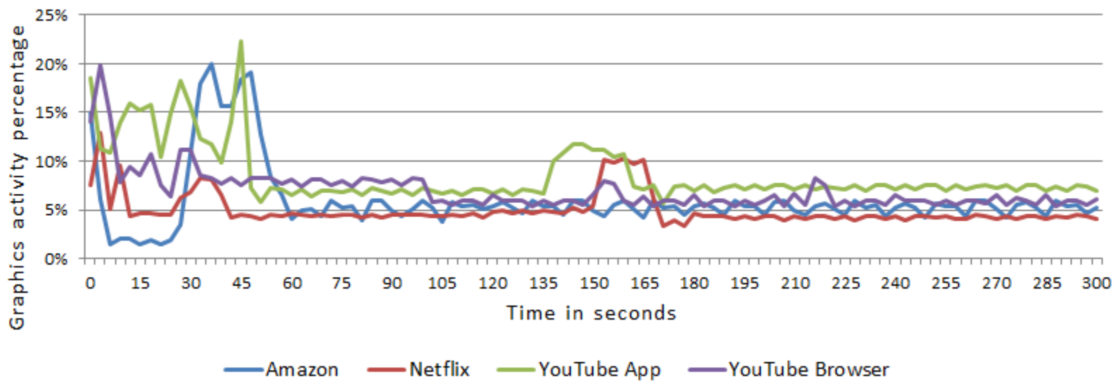


Figure 4.26: Graphics activity percentage collected on iPad Air 2 during video streaming scenario using Instrument.

Nexus 7 Video Streaming Results

We profiled the energy efficiency of the following video streaming apps: YouTube app, YouTube running in a browser, and Netflix app. Figure 4.27 represents the percentage of CPU utilization over time, and Table 4.6 represents the power metrics collecting using Trepn. We can rank the energy efficiency of video streaming apps as follows: YouTube app, Netflix app, and YouTube browser. In order to explain the results, we also examined the core C-states percentages in addition to the cores' frequency, as shown in Figures 4.31 and 4.30. We also examined the GPU load percentage in addition to its frequency, as shown in Figures 4.28 and 4.29.

YouTube app versus YouTube browser: First of all, in terms of GPU load percentage and GPU frequency, YouTube app and browser exhibited the exact same pattern, which completely overlapped on both of the graphs that represent the GPU metrics. As a result, we examined the remaining metrics. YouTube app had a higher CPU utilization percentage than YouTube browser, which cased the core to remain for 38.22% in 1512 MHz. Core 0 remained in active state for 52.87% of the time. On the other hand, YouTube running in a browser had less CPU utilization, which only triggered the CPU to remain in 1512 MHz for 23.77% but lead core 0 to remain active for 57.05% of the time. On the other hand, YouTube using a browser utilized more

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
YouTube App	1,280,419	1.91	1003.22	65	0
YouTube Browser	1,468,170	1.17	2146	77	1515
Netflix App	1,387,066	3.04	1023.78	65	0

Table 4.6: Power metrics collected on Nexus 7 during video streaming scenario using Trepn.

virtual memory than the app version, which confirmed the observation we noticed in the case of iPad (higher buffer percentage using a browser leading to more memory utilization compared to the app version.) Finally, all this information can explain the reason behind the differences in the energy efficiency of running YouTube via an app and via the browser.

YouTube app versus Netflix app: We also examined the differences between YouTube and Netflix. Both apps had very close numbers related to average virtual memory utilization, thread count, and total wakelocks. However, YouTube app had lower CPU percentage utilization and greater GPU load percentage compared to Netflix. That means that YouTube had better offloading from CPU to GPU algorithm compared to Netflix. Moreover, even though YouTube managed to have a higher GPU load compared to Netflix, it also managed to have lower GPU frequency than Netflix as well. The combination of all this information can explain why YouTube is more energy efficient than Netflix.

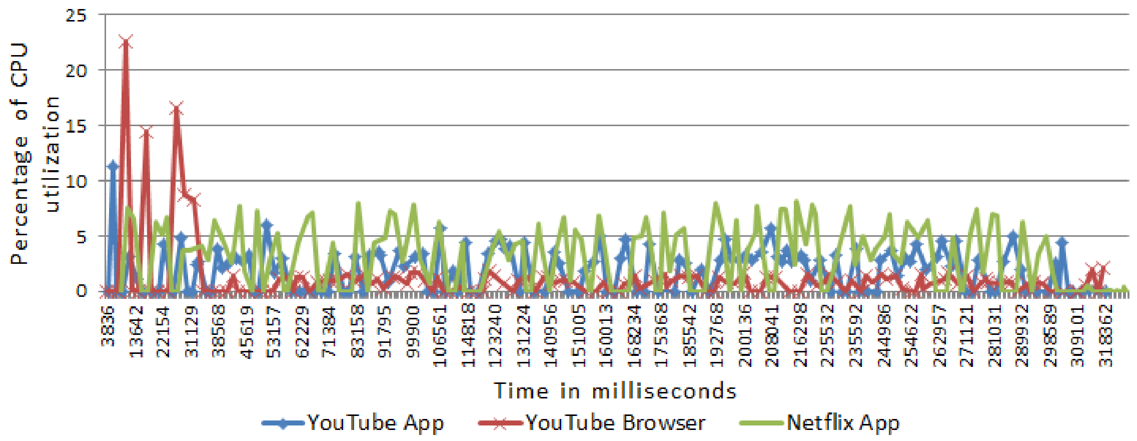


Figure 4.27: Percentage of CPU utilization collected on Nexus 7 during video streaming scenario using Trepn.

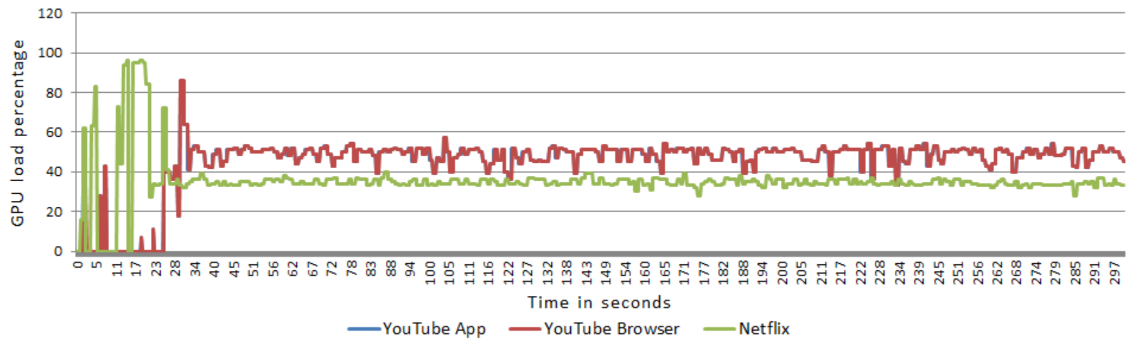


Figure 4.28: Percentage of GPU load collected on Nexus 7 during video streaming scenario using Trepn.

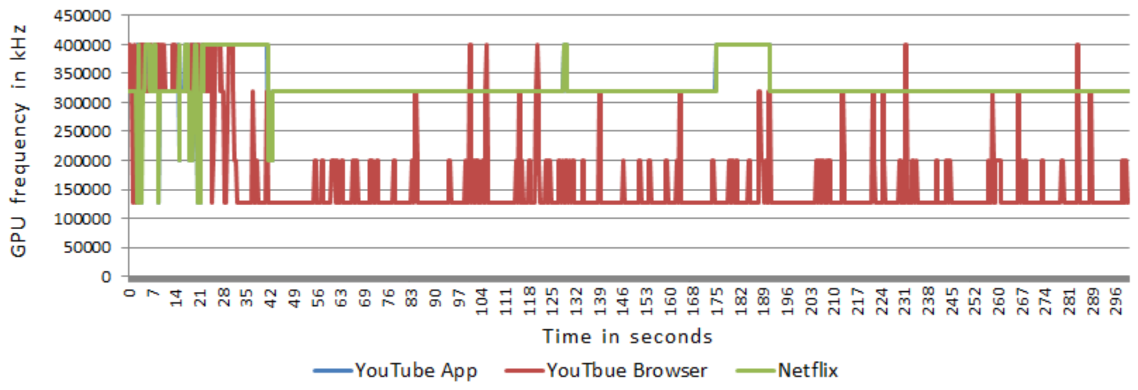


Figure 4.29: GPU frequency collected on Nexus 7 during video streaming scenario using Trepn.

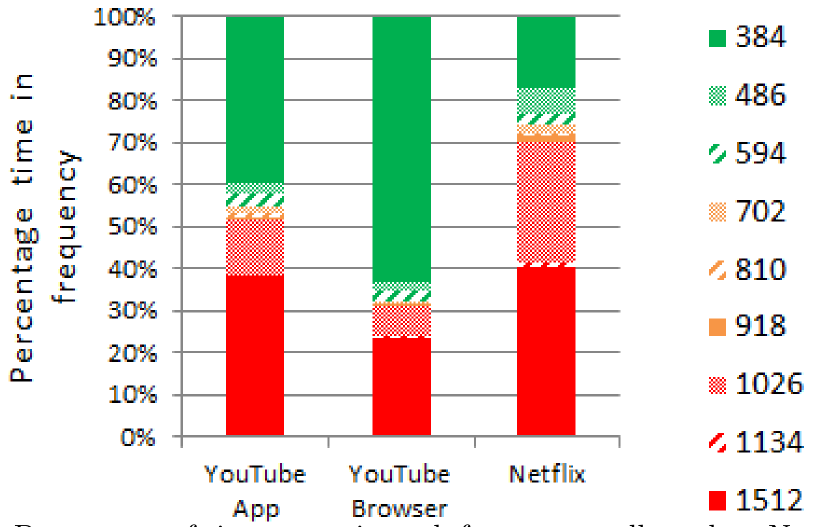


Figure 4.30: Percentage of time spent in each frequency collected on Nexus 7 during video streaming scenario using PowerMon.

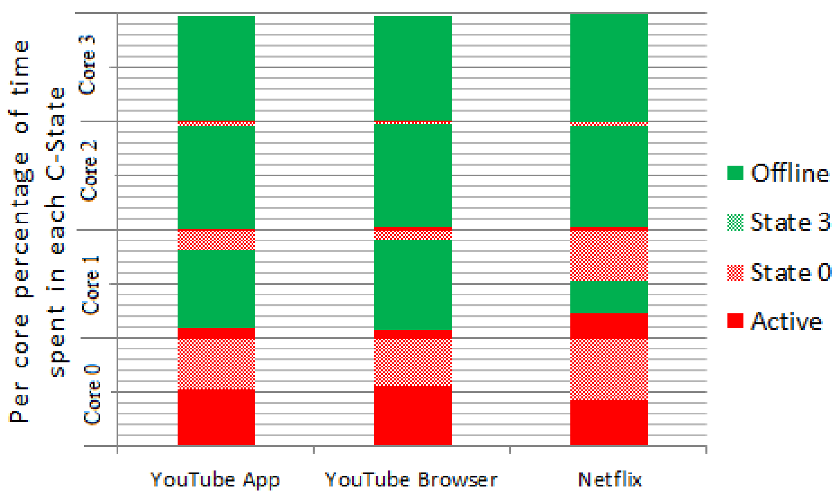


Figure 4.31: Percentage of time spent in each C-State per core collected on Nexus 7 during video streaming scenario using PowerMon.

Cross-Platform Comparison for Video Streaming Scenario

In order to compare the energy efficiency of video streaming apps cross platforms, we provide Table 4.7. We noticed that Netflix app ranked number 1 in energy efficiency on Surface 2 Pro and iPad Air, however, its ranking dropped to number 2 on Nexus 7, where YouTube app was ranked number 1.

Application	Surface 2 Pro	iPad Air	Nexus 7
Amazon	Rank 2	Rank 4	N/A
Amazon (browser)	Rank 5	N/A	N/A
Netflix	Rank 1	Rank 1	Rank 2
Netflix (browser)	Rank 4	N/A	N/A
YouTube	N/A	Rank 2	Rank 1
YouTube (browser)	Rank 3	Rank 3	Rank 3

Table 4.7: Cross platform video streaming apps energy efficiency ranking.

4.5.3 Music Streaming Scenario

Our third set of case studies are the music streaming scenarios, where we started streaming music using an app (or browser). Next, we started the profiling tool along with a 5-minute timer. Next, we relaunched the streaming app (browser) until the expiration of the timer. Then, we stopped the collection and saved the results.

Surface 2 Pro Music Streaming Results

During the music streaming scenario running on Surface 2 Pro, we selected three apps running using three different means: browser, desktop, and metro. By comparing the energy consumption as shown in Figure 4.32, we noticed that streaming music using metro style apps was the most energy efficient as opposed to streaming using a browser, which was the most energy inefficient. In order to explain the results, we examine core idle sleep states, package sleep states, frequency, hit count and total active duration, average wake-ups per package and per core, and timer resolution percentages, as shown in Figures 4.33, 4.34, 4.35, 4.36, 4.37, and 4.38, respectively.

XBOX versus Pandora and Spotify: We noticed that the percentage of core sleep states and package sleep states reflect the same pattern as energy consumption; however, XBOX metro app remained in greater percentage of time in high frequency. This means, that using XBOX metro, the performance is increased for a short duration in order to efficiently complete the task and allow the cores and package to go to sleep for a longer duration. On the other hand, the other two models (browser and desktop), remained in low performance state for a longer duration, resulting in longer duration in core and package active states.

Spotify versus Pandora: Moreover, we noticed that Spotify caused much more wake-ups to the core and package compared to Pandora, which could have potentially lead to much-higher energy consumption; however, since the total active time duration of Spotify is 25.21% less than Pandora, it offset the possibility of larger energy consumption. The large number of wake-ups caused by Spotify can be attributed to the application's change of the timer resolution interval from 15.6 ms to 1.0 ms for the majority of test duration.

Spotify versus XBOX: We also noticed that Spotify had seven different thread processes compared to two for XBOX. Using these results, since Spotify was only active for relatively short duration but with relatively high wake-ups, high number of threads, and short timer resolution interval, we conclude that Spotify can increase its energy efficiency by decreasing the number of wakeups through better synchronization of its threads and larger timer-resolution intervals.

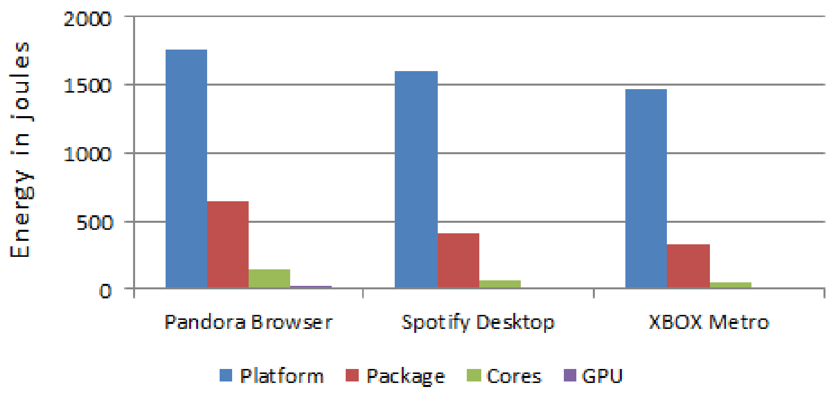


Figure 4.32: Energy consumed by Surface 2 Pro during music streaming scenario.

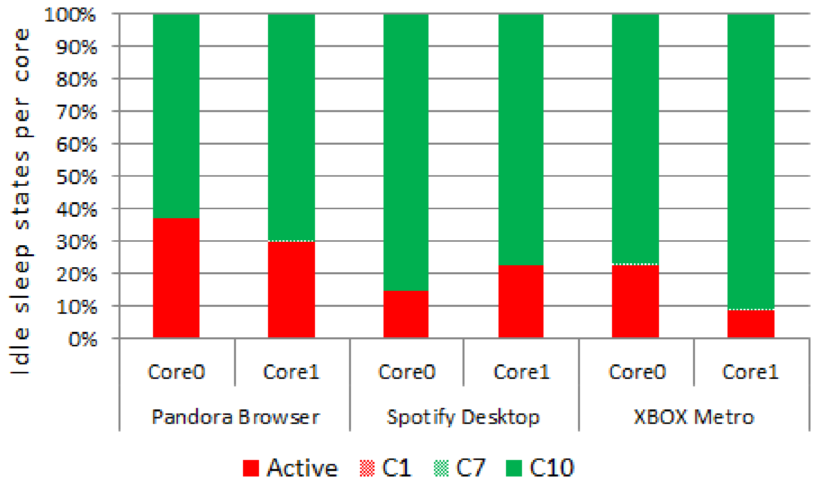


Figure 4.33: Idle sleep states percentage per core collected on Surface 2 Pro during music streaming scenario.

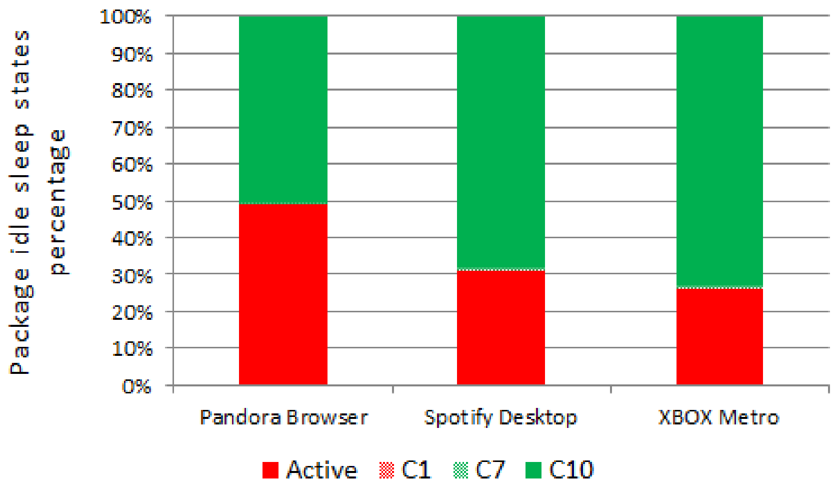


Figure 4.34: Package idle sleep states percentage collected on Surface 2 Pro during music streaming scenario.

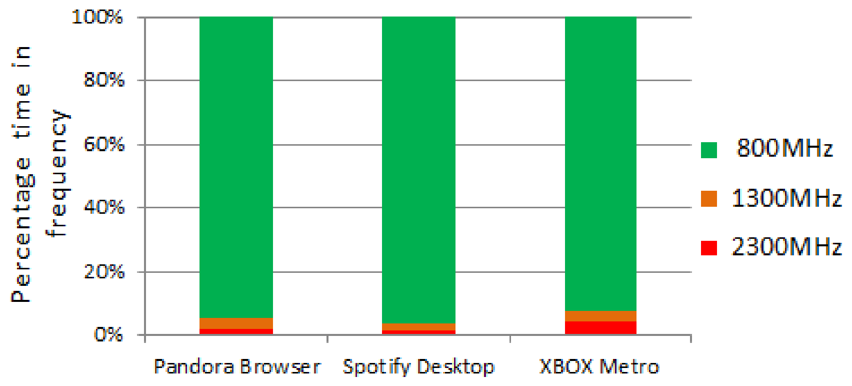


Figure 4.35: Core frequency distribution collected on Surface 2 Pro during music streaming scenario.

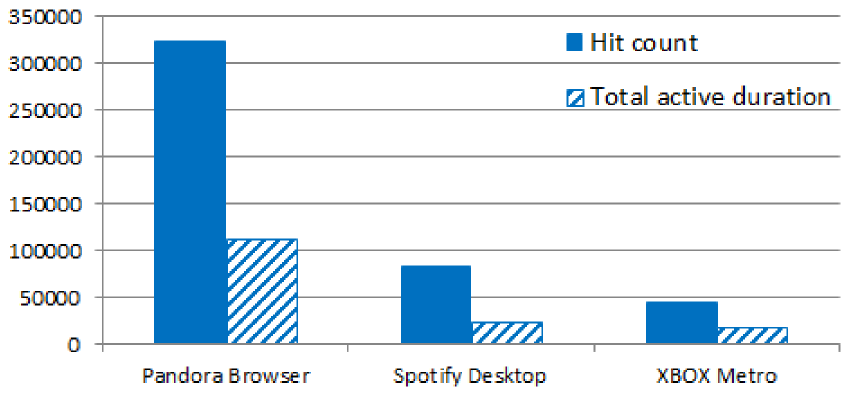


Figure 4.36: Total hit count and busy duration in milliseconds collected on Surface 2 Pro during music streaming scenario.

iPad Air Music Streaming Results

We profiled the energy efficiency of the following music streaming apps: Spotify, iTunes, and Pandora. Figure 4.39 displays the variation of energy consumption levels during the entire test duration. The average energy level is 5.35%, 3.50%, and 8.77% for Spotify, iTune, and Pandora, respectively. As a result, we can rank the energy efficiency of music streaming apps as follows: iTunes, Spotify, and Pandora. In order to examine the cause of the differences in energy consumption, we examined the total CPU activity percentages and graphics active percentages for all our test cases as shown in Figures 4.40 and 4.41, respectively.

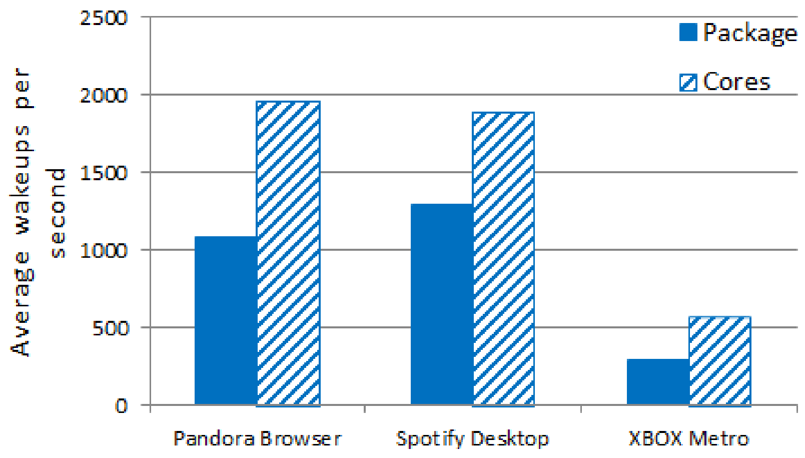


Figure 4.37: Average package and core wake-ups per seconds collected on Surface 2 Pro during music streaming scenario.

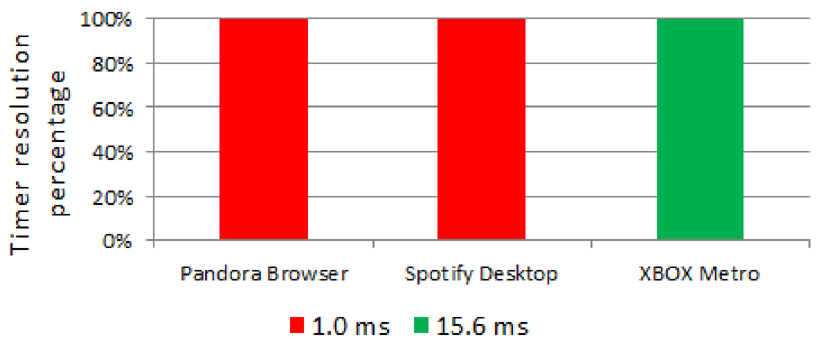


Figure 4.38: Percentage of time spent in each timer resolution interval.

Pandora versus Spotify versus iTunes: We noticed a reverse order of CPU utilization percentage compared to the app energy efficiency. This means that the most energy-efficient app has the highest percentage of CPU utilization, whereas the least energy-efficient app has the lowest percentage of CPU utilization. More specifically, the average CPU utilization of Spotify, iTunes, and Pandora are 9.22%, 13.01%, and 9.18%, respectively. Based on these results, the least energy-efficient app (Pandora) had the least CPU utilization, whereas, the most energy-efficient app (iTunes) was consuming the most CPU utilization. As a result, we examined the network activity patterns which revealed that iTunes had sent and received during long timer intervals large packets while at 2-second intervals received a small packet of 60 bytes. On the other hand, Pandora, sent out at regular 1-second intervals 166 bytes while sending and receiving during long timer intervals large packets similar to iTunes. Therefore, we can conclude that Pandora consumed more energy than iTunes because it kept the Wi-Fi radio at high power state for most of the test duration as opposed to iTunes which allowed the Wi-Fi radio to go to lower power states at regular intervals.

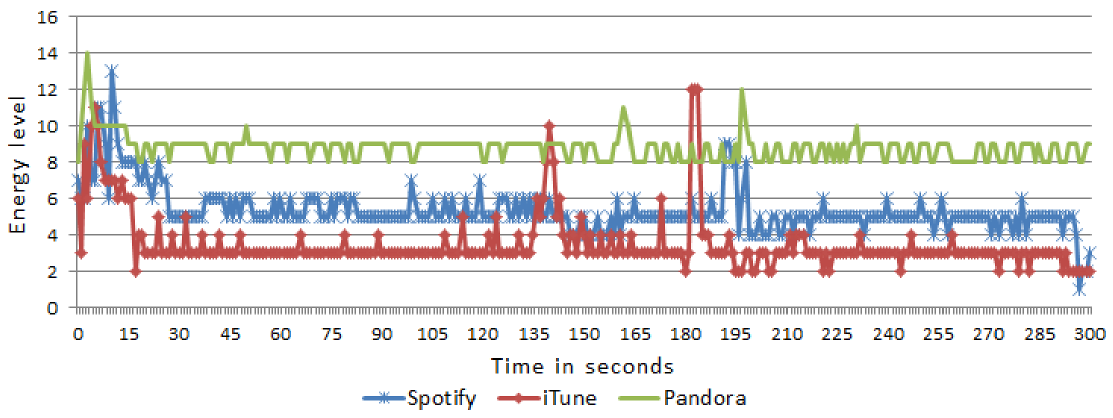


Figure 4.39: Energy level collected on iPad Air 2 during music streaming scenario using Instrument.

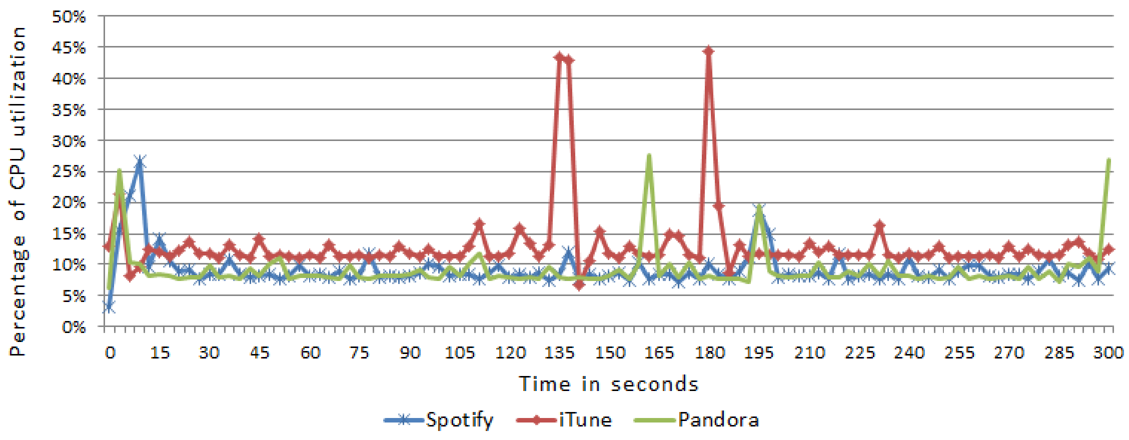


Figure 4.40: Total CPU activity percentage collected on iPad Air 2 during music streaming scenario using Instrument.

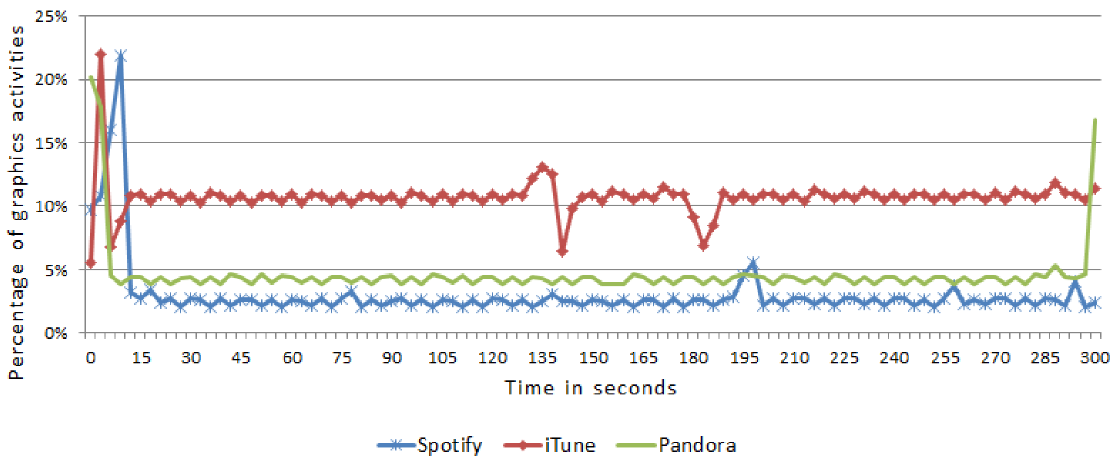


Figure 4.41: Graphics activity percentage collected on iPad Air 2 during music scenario using Instrument.

Nexus 7 Music Streaming Results

We profiled the energy efficiency of the following music streaming apps: Pandora, Spotify, and XBOX music. Figure 4.92 represents the percentage of CPU utilization over time, and Table 4.8 represents the power metrics collected using Trepn. Based on the results, we can rank the energy efficiency from the most energy efficient to the least energy efficient as XBOX, Spotify, and Pandora. In order to explain the results, we also examined the core C-states percentages in addition to the cores frequency, as shown in Figures 4.43 and 4.42.

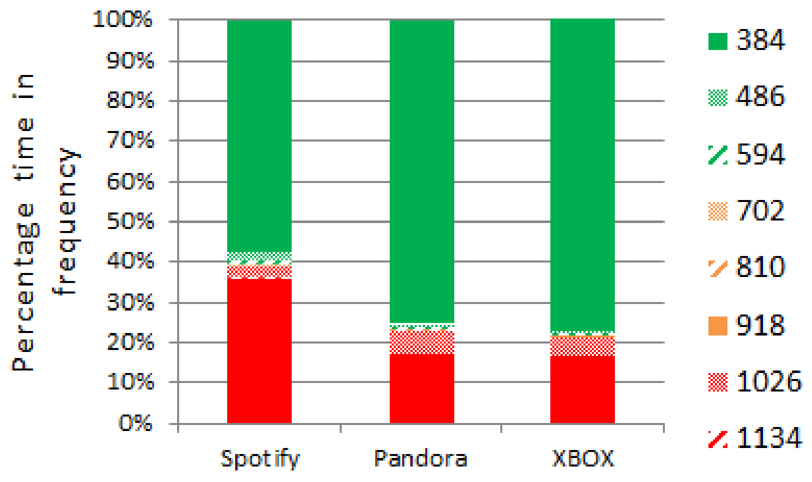


Figure 4.42: Percentage of time spent in each frequency collected on Nexus 7 during Music Streaming scenario using PowerMon.

Pandora: Even though Pandora had the least CPU usage, it kept the core in active state for the longest duration, which can be attributed to having the greatest number of wakelocks. In addition, it kept the CPU in high frequency for 39% of the duration which is around double the percentage than the other two apps.

XBOX versus Spotify: On the other hand, even though XBOX was the most energy efficient, music streaming had many interrupts, which we can attribute to the low average virtual memory usage compared to Spotify. The low average virtual memory means that XBOX experienced poor buffering of music. Therefore, XBOX sacrificed the user experience by optimizing the energy efficiency. Therefore, if we rank the combination of energy efficiency and user experience, Spotify takes the first spot, followed by Pandora, and followed by XBOX.

Cross-Platform Comparison for Music Streaming Scenario

In order to compare the energy efficiency of music streaming apps cross platforms, we provide Table 4.9. We noticed that on all platforms, Pandora was the least energy efficient app. In addition, even though XBOX is ranked number 1 on Nexus 7 and

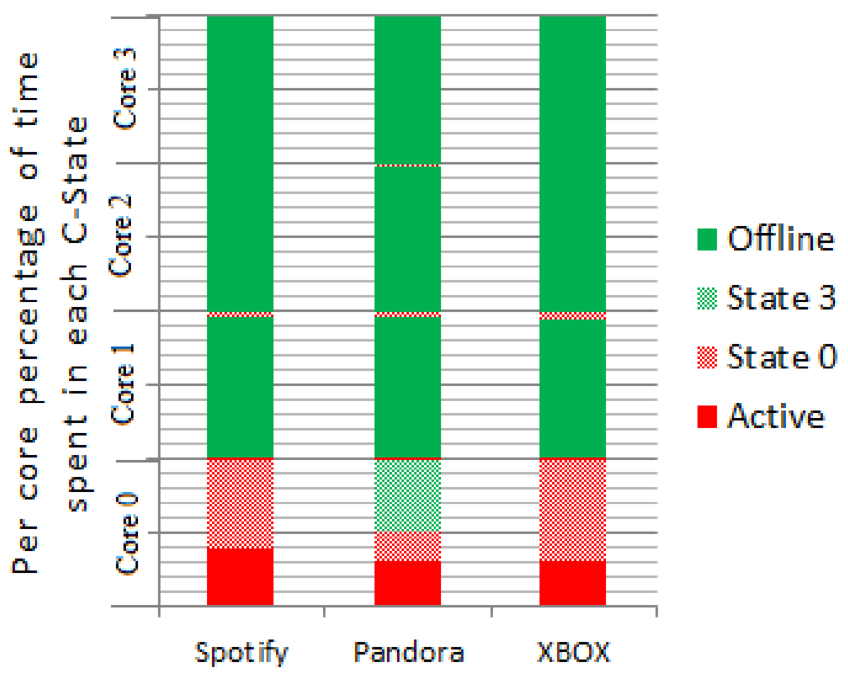


Figure 4.43: Percentage of time spent in each C-State per core collected on Nexus 7 during music streaming scenario using PowerMon.

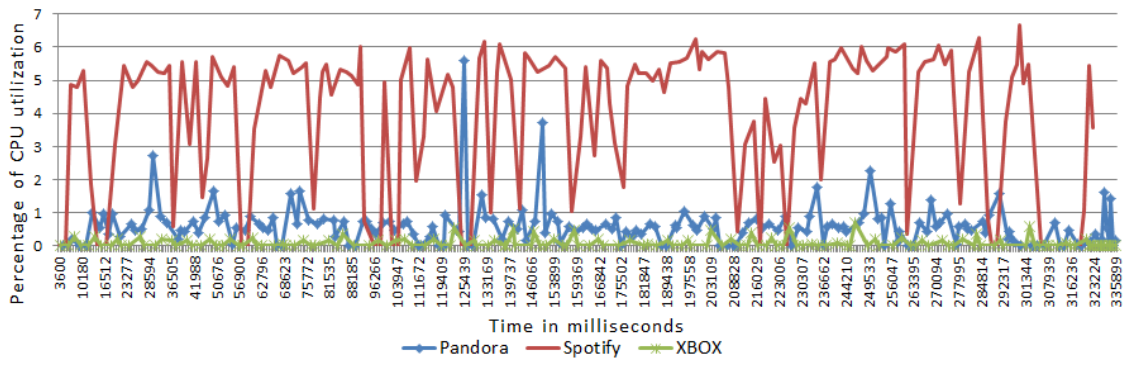


Figure 4.44: Percentage of CPU utilization collected on Nexus 7 during music streaming scenario using Trepn.

App Name	**Average Power in uW**	**Average CPU Percentage**	**Average Virtual Memory**	**Number of Threads**	**Total wake-locks**
Pandora	705,156	0.59	973.38	42	1714
Spotify	683,714	4.01	1860.03	107	1646
XBOX	483,612	2.08	981.23	45	1550

Table 4.8: Power metrics collected on Nexus 7 during music streaming scenario using Trepn.

Application	**Surface 2 Pro**	**iPad Air**	**Nexus 7**
Pandora	N/A	Rank 3	Rank 3
Pandora (browser)	Rank 3	N/A	N/A
Spotify	Rank 2	Rank 2	Rank 2
XBOX	Rank 1	N/A	Rank 1
iTune	N/A	Rank 1	N/A

Table 4.9: Cross platform music streaming apps energy efficiency ranking.

Surface 2 Pro, XBOX rank needs to be downgraded in the case of Nexus 7 because XBOX had quality of service (QoS) issues.

4.5.4 Map Scenario

Our fourth set of case studies are the map scenarios where we started the profiling tool along with a 3-minute timer, launched the mapping app, typed an address (same for all apps), started routing, and kept the screen on until the expiration of the timer. Then, we stopped the collection and saved the results.

Surface 2 Pro Map Results

We tried running two different apps on Surface 2 Pro, which are Google Maps and Windows Maps. However, we were not able to start any routes. As a result, we disregarded all the maps results for this platform since they did not match the collection on the other two platforms.

iPad Air Map Results

We profiled the energy efficiency of the following map apps: Apple Maps, Google Maps, and Waze. Figure 4.45 displays the variation of energy consumption levels during the entire test duration. The average energy level is 8.28%, 8.72%, and 8.59%, respectively. As a result, we can rank the energy efficiency of map apps as follows: Apple Maps, Waze, and Google Maps. In order to examine the cause of the differences in energy consumption, we examine the total CPU and graphics activity percentages for all our test cases, as shown in Figures 4.46 and 4.47, respectively.

Google maps versus Apple maps: Similar to the music streaming case scenarios, we noticed that the most energy-efficient apps also had the highest CPU and graphics utilization percentages, whereas the least energy-efficient apps had the lowest CPU and graphics utilization percentages. As a result, we examined the network activity pattern. We noticed that in the case of Google Maps, at an approximate interval of 5 seconds, there is an average of 17,149 bytes received and an average of 2,248 bytes sent, and at about 2-second intervals, there is an average of 60 bytes received. Also, at approximately 40-second intervals, there is an average of 859 bytes received and 288 bytes sent. On the other hand, Apple maps had a consecutive packets sent and received during 5 seconds interval of medium size of an average 180 bytes followed by a duration of approximately 40 seconds with absolutely no packets sent or received. Then, again, a busy 5-second duration where medium size packets are sent and received, followed by the 40 seconds of no activities. This pattern was repeated for the entire test duration. These network activity patterns can explain why Apple maps was the most energy efficient even though it had the highest CPU and graphics utilization. Apple maps enabled the Wi-Fi radio to go to deep sleep states for much longer than Google maps. As a result, it reduced the energy consumption of the Wi-Fi radio resulting in reduction of the entire platform power consumption despite the increase in CPU and graphics utilization.

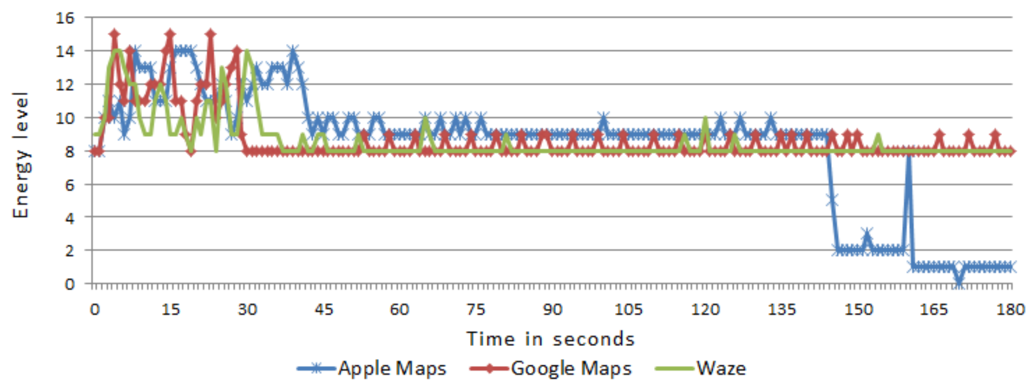


Figure 4.45: Energy level collected on iPad Air 2 during map scenario using Instrument.

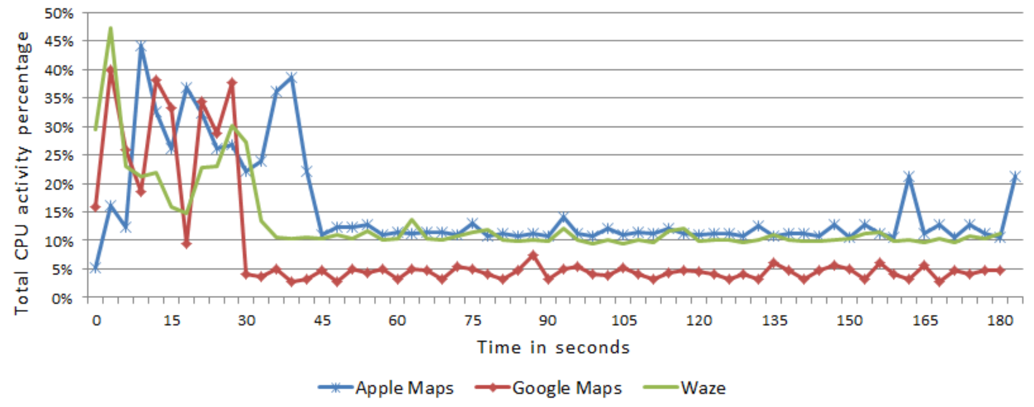


Figure 4.46: Total CPU activity percentage collected on iPad Air 2 during map scenario using Instrument.

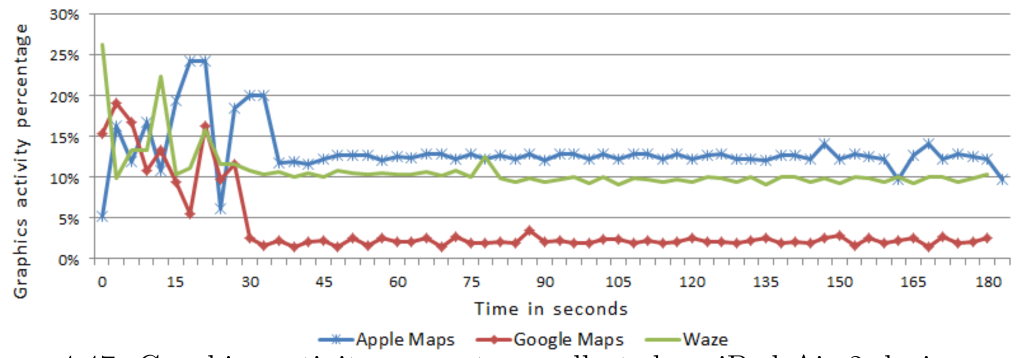


Figure 4.47: Graphics activity percentage collected on iPad Air 2 during map scenario using Instrument.

Nexus 7 Map Results

We profiled the energy efficiency of the following mapping apps: Waze and Google Maps. Figures 4.50, 4.49, and 4.48 represent the percentage of CPU utilization over time, core C-states, and frequency, respectively. Table 4.10 represents the power metrics collected using Trepn. Based on the results, Waze is more energy efficiently than Google Maps. As a matter of fact, Google maps consumed on average more than double the power consumed by Waze. This is attributed to the fact that Waze had less percentage of active C-States and less percentage of high frequency compared to Google Maps.

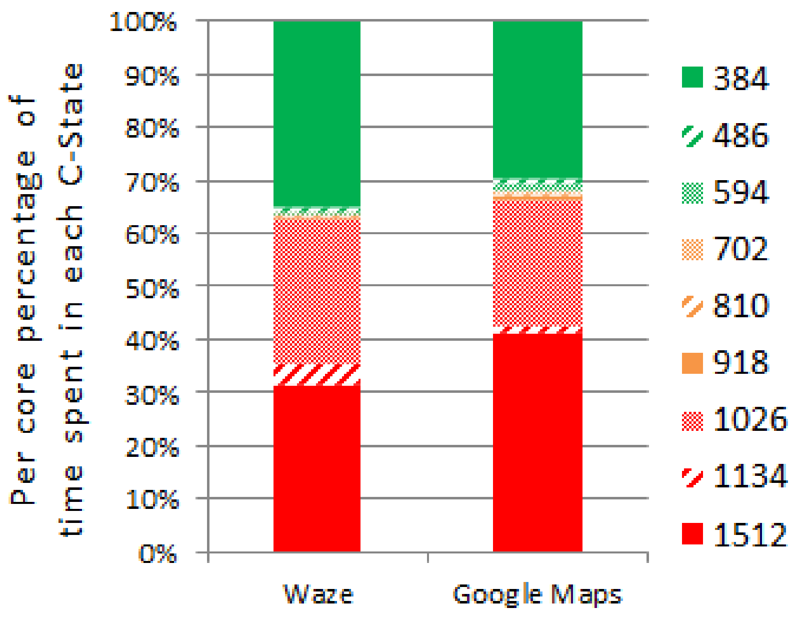


Figure 4.48: Percentage of time spent in each frequency collected on Nexus 7 during Map scenario using PowerMon.

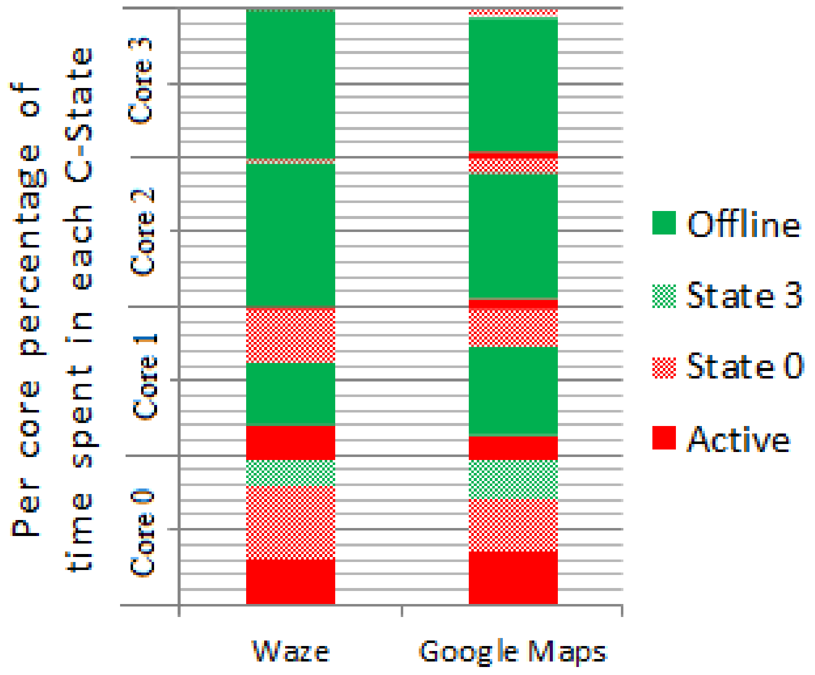


Figure 4.49: Percentage of time spent in each C-State per core collected on Nexus 7 during Map scenario using PowerMon.

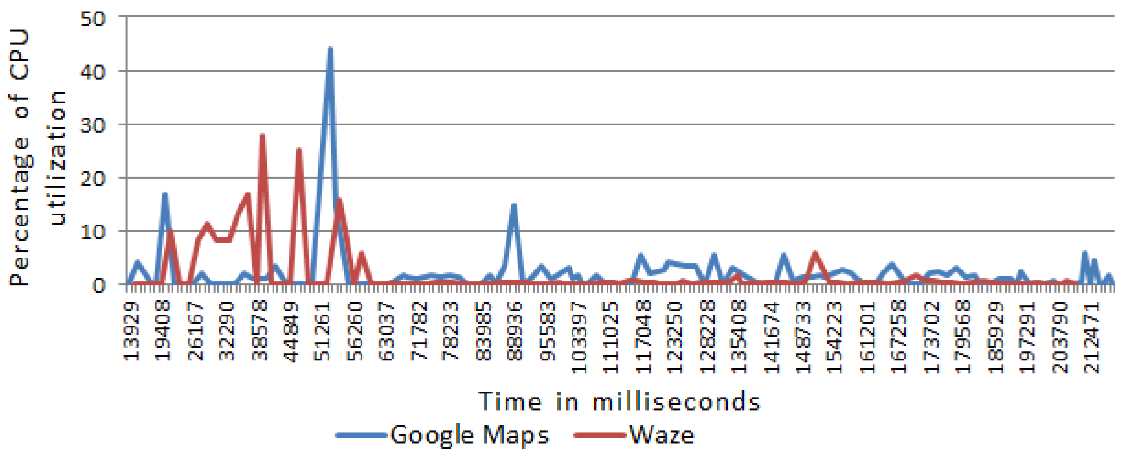


Figure 4.50: Percentage of CPU utilization collected on Nexus 7 during map scenario using Trepn.

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
Waze	243,119	1.82	1033.47	28	1
Google Maps	656,168	2.09	1002.02	50	1218

Table 4.10: Power metrics collected on Nexus 7 during map scenario using Trepn.

Application	iPad Air	Nexus 7
Apple Maps	Rank 1	N/A
Waze	Rank 2	Rank 1
Google Maps	Rank 3	Rank 2

Table 4.11: Cross-platform map apps energy efficiency ranking.

Cross-Platform Comparison for Map Scenario

In order to compare the energy efficiency of map apps cross platforms, we provide Table 4.11. We noticed that Google Maps was the least energy-efficient app on both iPad and Nexus 7. In addition, Waze ranked higher on the energy efficiency scale compared to Google Maps.

4.5.5 Video Chatting Scenario

Our fifth set of case studies are the video chatting scenarios, where we started the power profiling tool along with a 5-minute timer, launched the video chatting app, initiated an invite for a video chatting session, and had a second user accept the invite. Then, upon the timer expiration, we stopped profiling and saved the results.

Surface 2 Pro Video Chatting Results

During our experiment for video chatting scenarios on Surface 2 Pro, we used Skype as a metro app, and Hangout, a metro app which upon launching a video chatting session, launches a browser. By comparing the results, we noticed that hangout consumed more energy than Skype, as shown in Figure 4.51. In order to explain the results, we examined core idle sleep state, package sleep states, frequency, and average wake-ups per package and per core, as shown in Figures 4.52, 4.53, 4.54, and 4.55, respectively.

Skype vs Hangout: We noticed that both cores and package remained for longer duration in active states when comparing Hangout to Skype. Similarly, Hangout caused the cores to remain in high performance states for the majority of the test duration, unlike Skype. For instance, Skye remained for 79.96% and 3.48% in 800 MHz and 2300 MHz, respectively, whereas Hangout remained for 10.73% and 51.53% in 800 MHz and 2300 MHz, respectively. Finally, average wakeups per package for Skype and Hangout as shown in Figure 4.55 are similar. These values are not surprising even though Skype and Hangout consume different energy. The average wakeups are lower than expected in the case of Hangout, which can be attributed to the greater percentage of active package, which resulted in less percentage of sleep time, and thus less opportunity to wake the package and core. Finally, based on our previous test case scenarios, we can predict that the energy efficiency of Hangout can be increased if the app is implemented to be completely native due to the expected higher energy consumption values when comparing native to browser based apps.

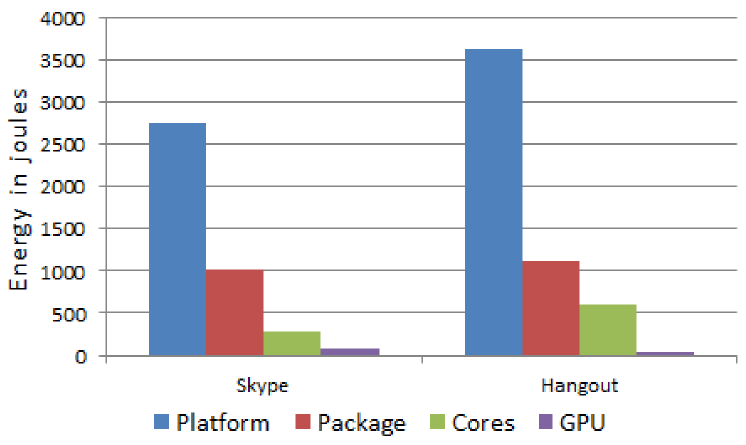


Figure 4.51: Energy consumed by Surface 2 Pro during video chatting scenario.

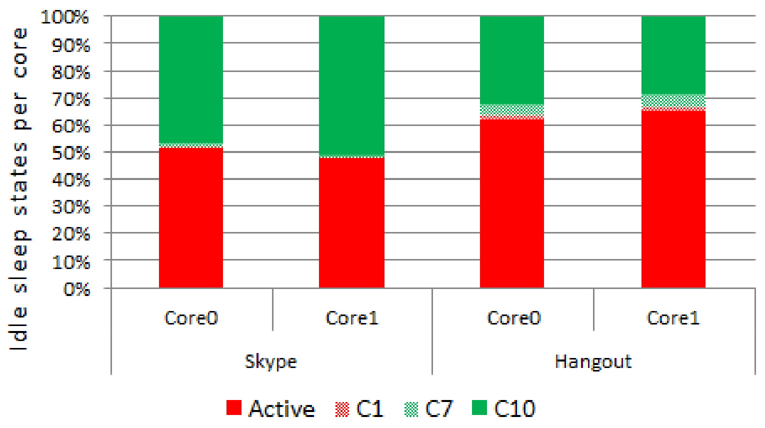


Figure 4.52: Idle sleep states percentage per core collected on Surface 2 Pro during video chatting scenario.

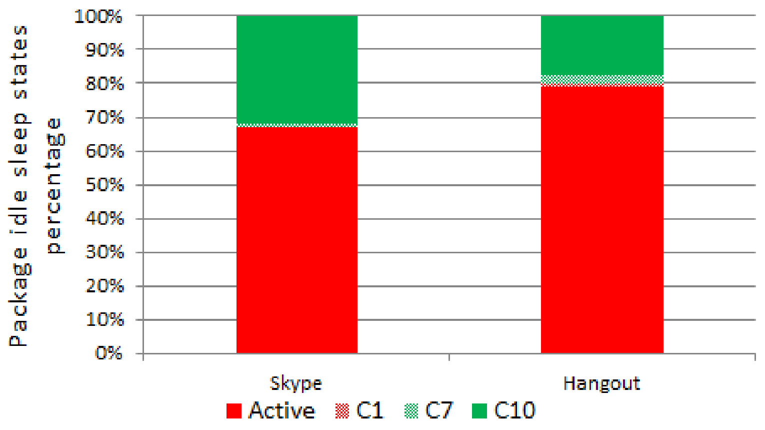


Figure 4.53: Package idle sleep states percentage collected on Surface 2 Pro during video chatting scenario.

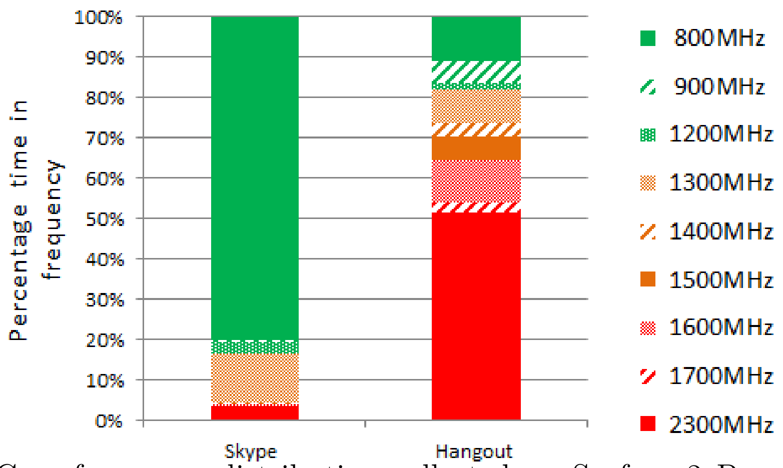


Figure 4.54: Core frequency distribution collected on Surface 2 Pro during video chatting scenario.

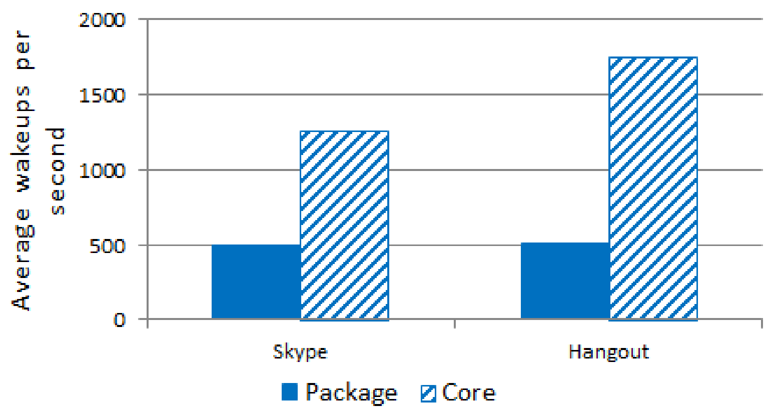


Figure 4.55: Average package and core wake-ups per seconds collected on Surface 2 Pro during video chatting scenario.

iPad Air Video Chatting Results

We profiled the energy efficiency of the following video chatting apps: Hangout and Skype. Figure 4.56 displays the variation of energy consumption levels during the entire test duration. The average energy level is: 14.56 and 14.95 for Hangout and Skype, respectively. Thus, Hangout is more energy efficient than Skype. In order to examine the cause of the difference in energy consumption, we examined the total CPU activity percentages as shown in Figure 4.57, which showed an average of 40.25% and 57.75% for Hangout and Skype, which support the differencse in energy consumption.

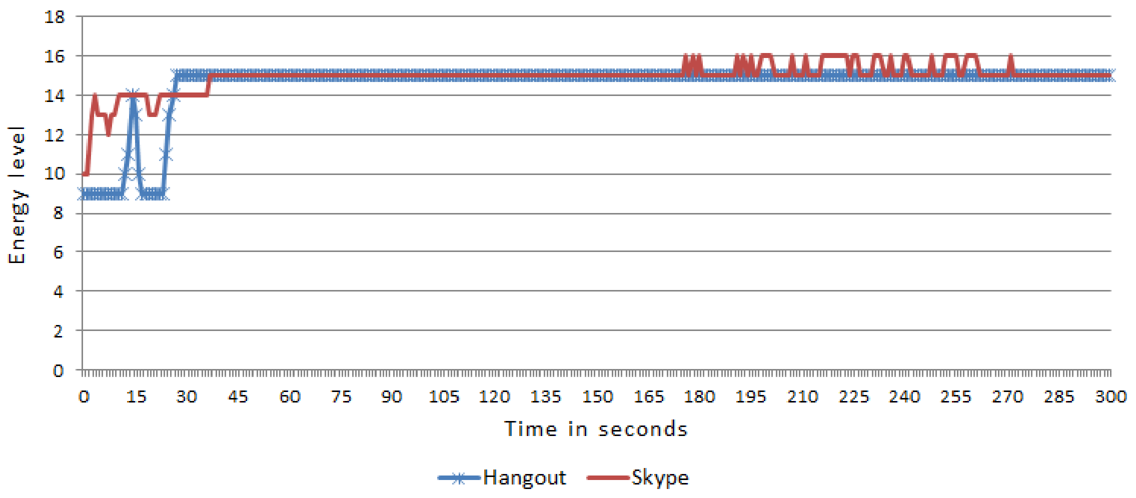


Figure 4.56: Energy level collected on iPad Air 2 during video chatting scenario using Instrument.

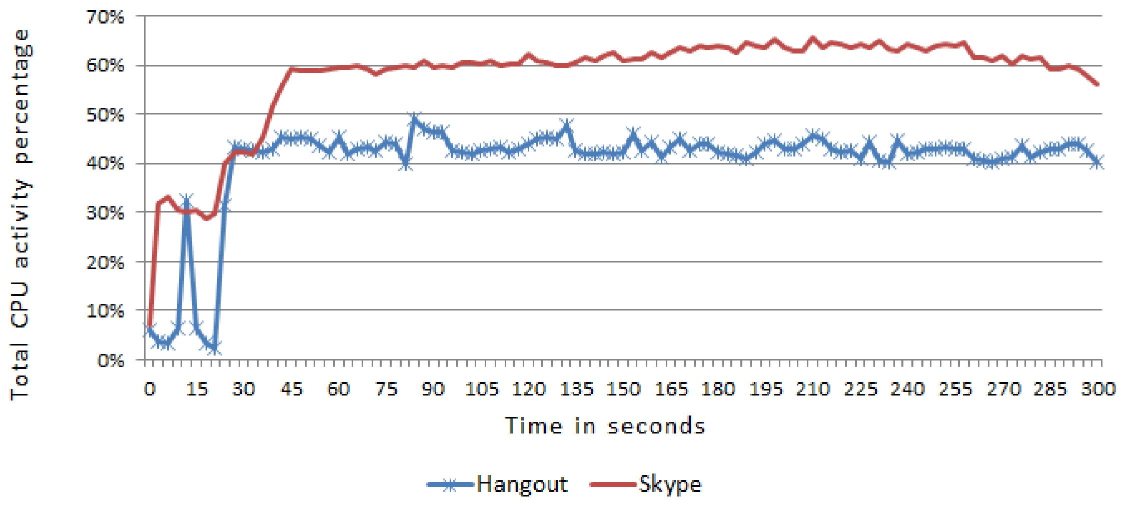


Figure 4.57: Total CPU activity percentage collected on iPad Air 2 during video chatting scenario using Instrument.

Nexus 7 Video Chatting Results

We profiled the energy efficiency of the following video chatting apps: Skype and Hangout. Figures 4.58, 4.59, and 4.60 represent the percentage of CPU utilization over time, core C-states, and frequency, respectively. Table 4.12 represents the power metrics collected using Trepn. Based on the results, Skype is much more energy efficient than Hangouts, which is reflected in the average CPU utilization, average virtual memory, total wakeups, percentage of high frequency, and percentage of active cores.

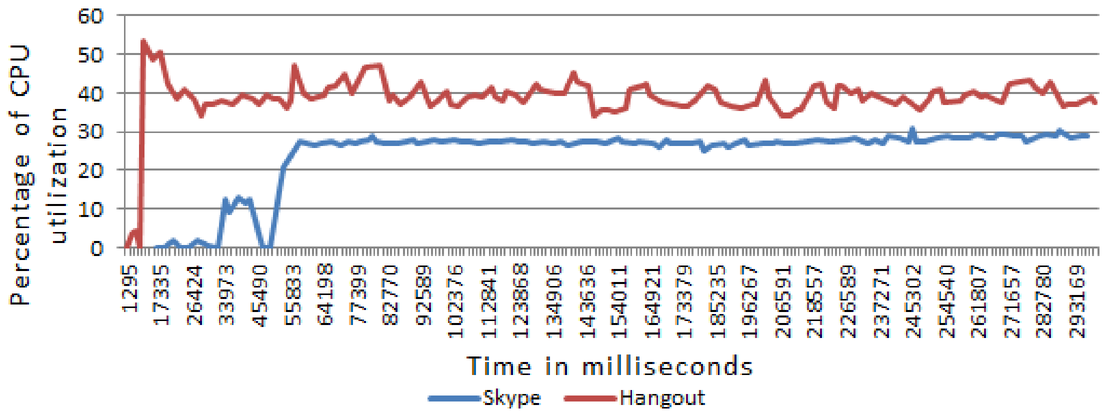


Figure 4.58: Percentage of CPU utilization collected on Nexus 7 during browser scenario using Trepn.

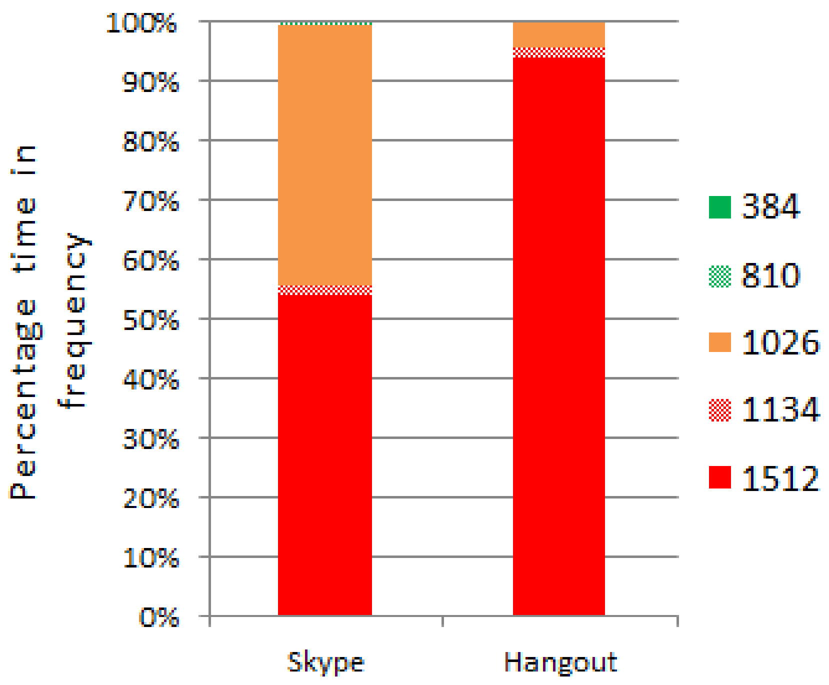


Figure 4.59: Percentage of time spent in each frequency collected on Nexus 7 during video chatting scenario using PowerMon.

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
Skype	2,155,302	23.13	1000.56	40	1138
Hangout	3,184,398	34.65	1017.73	50	1441

Table 4.12: Power metrics collected on Nexus 7 during video chatting scenario using Trepn.

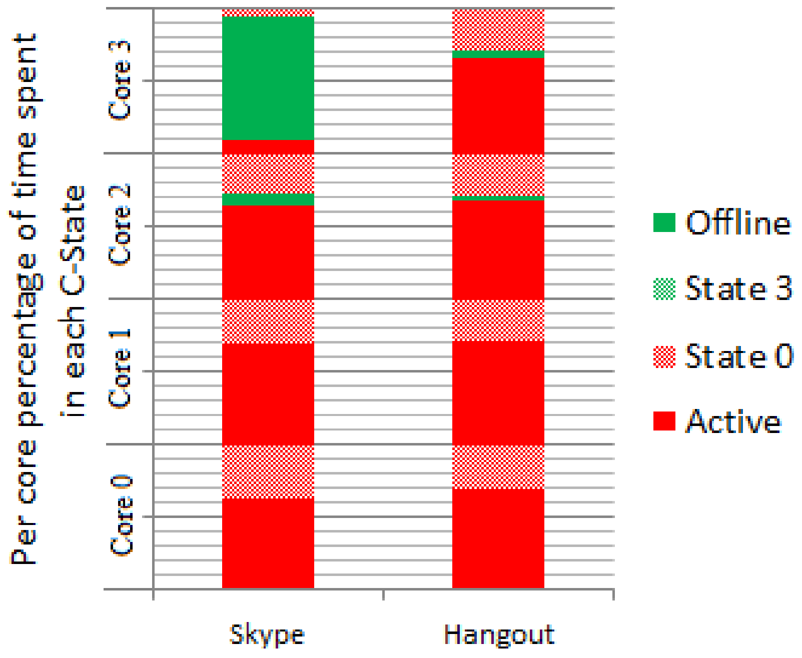


Figure 4.60: Percentage of time spent in each C-State per core collected on Nexus 7 during video chatting scenario using PowerMon.

Cross-Platform Comparison for Video Chatting Scenario

In order to compare the energy efficiency of video chatting apps cross platforms, we provide Table 4.13. We found out that Skype was 2:1 more energy efficient than Hangout.

Application	Surface 2 Pro	iPad Air	Nexus 7
Hangouts	Rank 2	Rank 1	Rank 2
Skype	Rank 1	Rank 2	Rank 2

Table 4.13: Cross-platform video chatting energy efficiency ranking

4.5.6 Cloud Storage Scenario

During cloud storage scenario, we started the profiling tool along with the 3-minute timer. Next, we launched the cloud storage app and kept the screen on. Upon the expiration of the timer, we stopped the collection and saved the results. In the case of accessing the cloud storage via a browser, we first launched the browser and typed the user name and password. Then, we started the profiling tool along with the 3-minute timer. Next, we launched the browser and clicked on sign in. We kept the screen on until the timer expired. Next, we stopped the collection and saved the results. Please note that all cloud storage accounts stored the same data size.

Surface 2 Pro Cloud Storage Results

We profiled the energy efficiency of the following: Dropbox metro, Dropbox accessed through a browser, Google Drive accessed through a browser, and SkyDrive metro. Based on the energy consumption values as shown in Figure 4.61, we can rank them from the most energy efficient to the least as follows: SkyDrive, Dropbox metro, Google Drive browser, and Dropbox browser. In order to examine the cause of the difference in energy consumption, we examine core idle sleep states, package sleep states, number of wakeups, total hit count and active duration, average wake-ups per second for package and cores, and timer-resolution, as shown in Figures 4.62, 4.63, 4.64, 4.65, 4.66, 4.65, and 4.67, respectively.

Google Drive browser vs Dropbox browser: We noticed that the percentage of active core and package C-States reflects the energy efficiency order of apps where the most energy-efficient apps were active for the least time percentage and vice versa. Likewise, the percentage of high frequency and total active duration reflect the application energy-efficiency order. However, the average number of wake-ups per second in the case of DropBox accessed via a browser is less than Google Drive, which is due to the fact that the Dropbox accessed via a browser was active for longer duration, thus the platform didn't go as often to sleep. As a result, there were fewer opportunities to wake the package and core up because they were already active.

Dropbox and Skydrive versus Google Drive and Dropbox browser: DropBox metro and SkyDrive did not change the timer resolution as the other two apps, which changed it to 1.0 ms. As a result, they enabled the package and core to remain in idle sleep states for longer duration than Dropbox browser, and Google Drive, leading them to be more energy efficient than the later two.

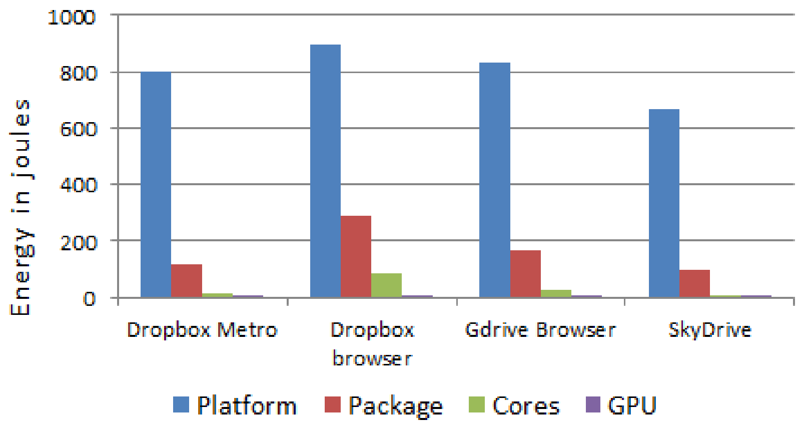


Figure 4.61: Energy consumed by Surface 2 Pro during cloud storage scenario.

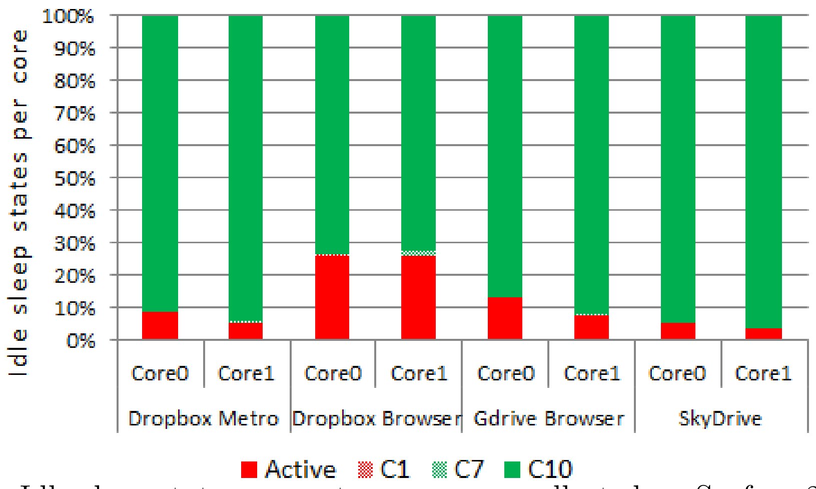


Figure 4.62: Idle sleep states percentage per core collected on Surface 2 Pro during cloud storage scenario.

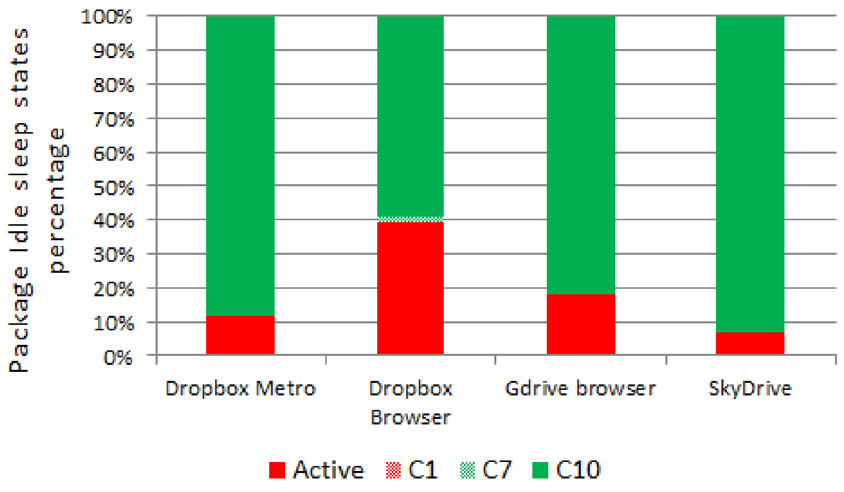


Figure 4.63: Package idle sleep states percentage collected on Surface 2 Pro during cloud storage scenario.

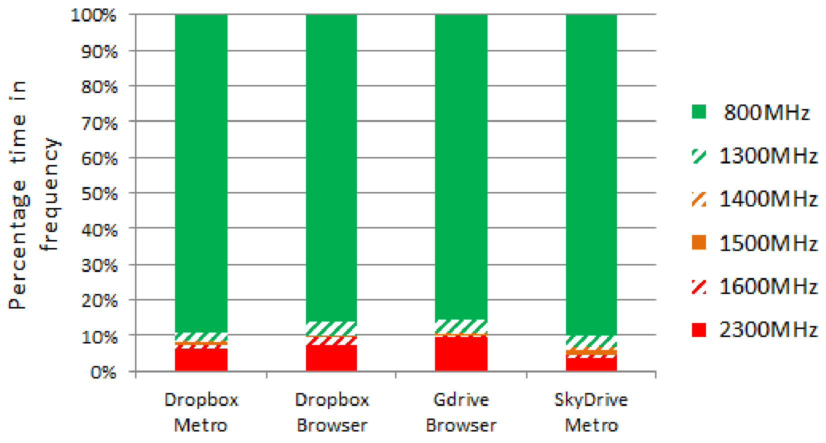


Figure 4.64: Core frequency distribution collected on Surface 2 Pro during cloud storage scenario.

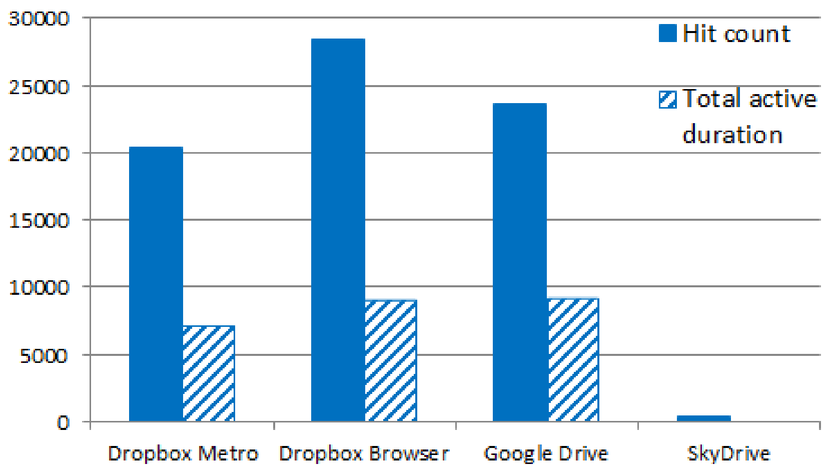


Figure 4.65: Total hit count and busy duration in milliseconds collected on Surface 2 Pro during cloud storage scenario.

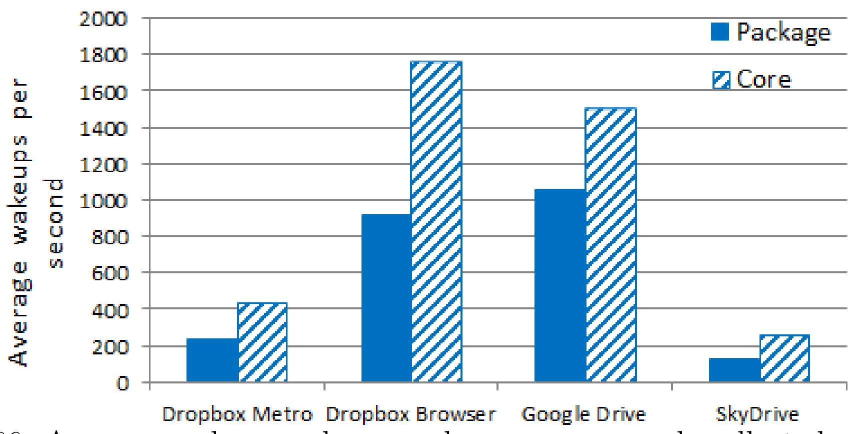


Figure 4.66: Average package and core wake-ups per seconds collected on Surface 2 Pro during cloud storage scenario.

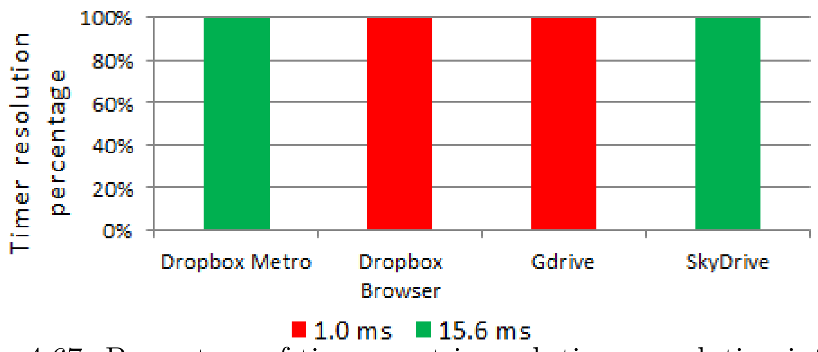


Figure 4.67: Percentage of time spent in each timer resolution interval.

iPad Air Cloud Storage Results

We profiled the energy efficiency of the following cloud storage cases: SkyDrive, Dropbox accessed via a browser, Dropbox app, Google Drive accessed via a browser, and Google drive app. Figure 4.68 displays the variation of energy consumption levels during the entire test duration. The average energy level is: 1.93, 9.15, 11.02, 8.39, and 8.42 for SkyDrive, Dropbox browser, Dropbox app, Google Drive browser, and Google Drive app, respectively. Based on the results, we can rank the energy efficiency of cloud storage apps as follows: SkyDrive, Google Drive App, Google Drive browser, Dropbox browser, and Dropbox app. In order to examine the cause of the differences in energy consumption, we examined the total CPU utilization and graphics activity percentages for all our test cases, as shown in Figure 4.69 and 4.70, respectively. The average CPU utilization is 3.88%, 9.81%, 9.71%, 3.97%, and 4.38%, and average graphics activity is 2.44%, 5.09%, 6.42%, 2.73%, and 2.45% for SkyDrive, Dropbox browser, Dropbox app, Google Drive browser, and Google drive app, respectively.

Dropbox app vs Dropbox browser: The only contradicting observation between this case scenario and all previous scenarios (where apps were more energy efficient than accessing content using a browser), is that Dropbox browser is more energy efficient than Dropbox app. By analyzing the results, we found out that average CPU utilization percentages still align with our previous observations where browser version utilized more CPU time, however, the app version had a sophisticated user interface leading to higher average graphics utilization percentage. Thus, it required more energy than the browser version for rendering and reduced the overall energy efficiency.

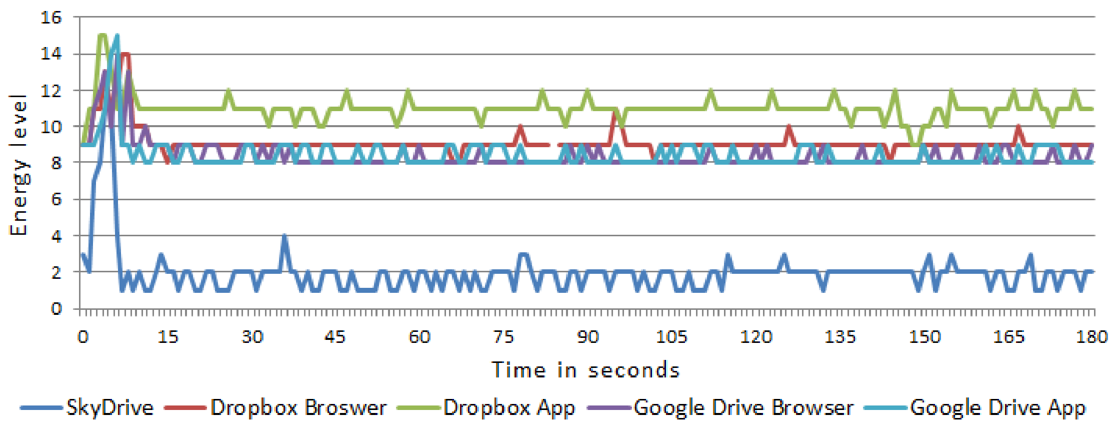


Figure 4.68: Energy level collected on iPad Air 2 during cloud storage scenario using Instrument.

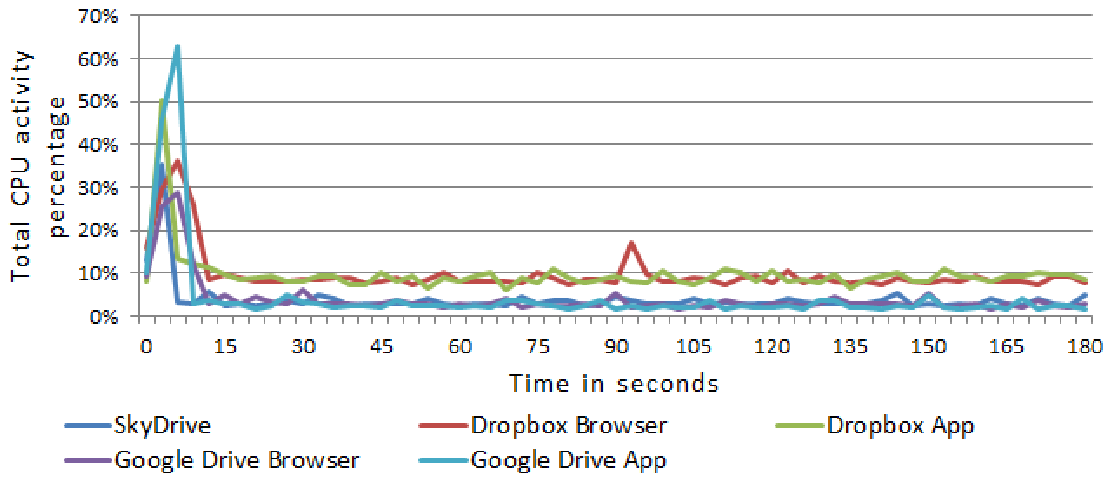


Figure 4.69: Total CPU activity percentage collected on iPad Air 2 during cloud storage scenario using Instrument.

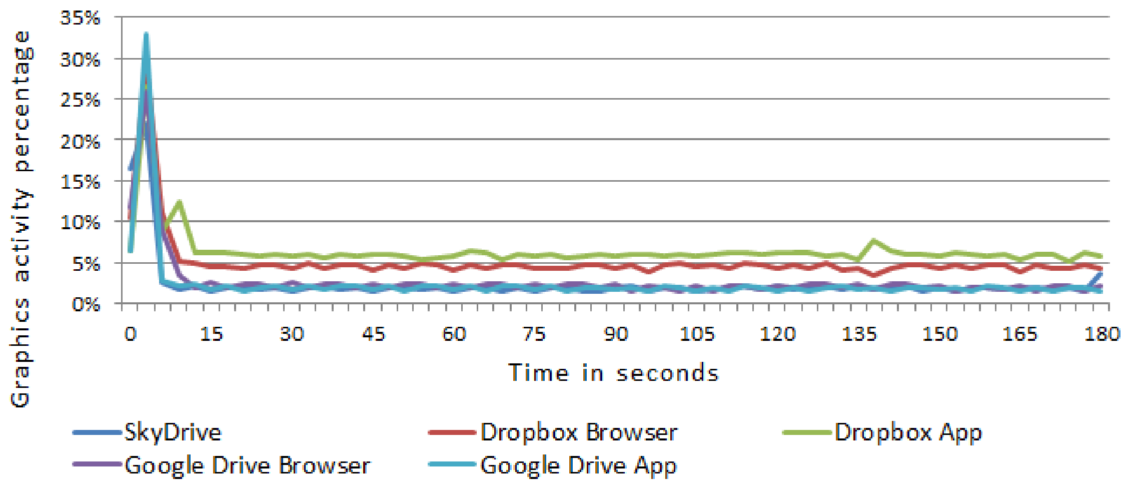


Figure 4.70: Graphics activity percentage collected on iPad Air 2 during cloud storage scenario using Instrument.

Nexus 7 Cloud Storage Results

We profiled the energy efficiency of the following cloud storage apps: Google Drive app, SkyDrive app, Dropbox app, and Dropbox accessed using a browser. Figures 4.72 and 4.71 represent the percentage of core C-states and frequency, respectively. Table 4.14 represents the power metrics collected using Trepn. Based on the results, we can rank the energy efficient of cloud storage apps as follows: Dropbox App, Dropbox browser, Google Drive app, and SkyDrive.

Dropbox app versus Dropbox browser: Dropbox using a browser consumed 5 times more average power than the app version, which can be attributed to the presence of wakelocks in the case of browser. In addition, the app version had lower percentage of CPU utilization, which led to lower active core percentage and extended duration in 384 MHz compared to the browser version.

Google Drive app versus Dropbox app: By examining the percentage of CPU utilization, average virtual memory utilization, and number of threads, we noticed that the values for the listed metrics are similar to both apps. However, there is a huge difference in the average power consumption. As a result, we examined the core frequency and noticed that in the case of Google Drive, the CPU remained in high frequency (1512 and 1026 MHz) for longer duration than Dropbox app. We also examined the remaining metrics from Trepn, and noticed that Google Drive acquired a wifilock for 25 seconds. That means for the entire duration (25 seconds), the Wi-Fi radio remained in high power state. Thus, this explains why with similar metrics (CPU, memory, and thread), Google Drive still consumed significantly more energy than Dropbox.

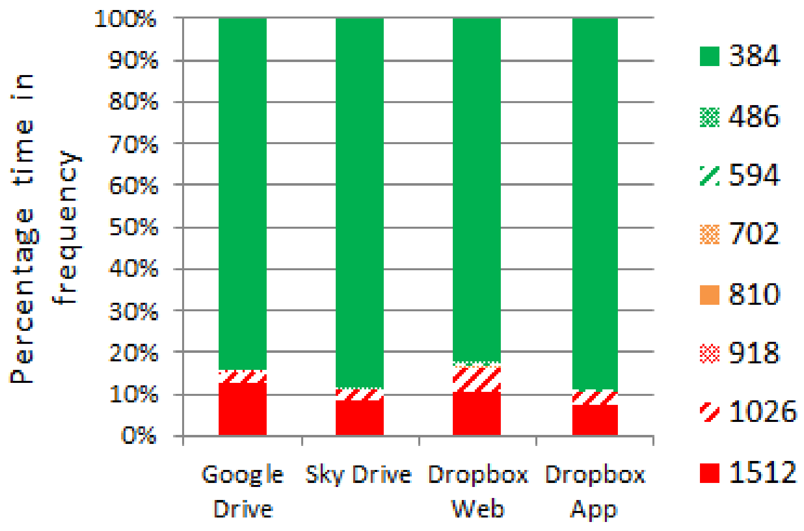


Figure 4.71: Percentage of time spent in each frequency collected on Nexus 7 during cloud storage scenario using PowerMon.

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
Google Drive App	701,320	0.28	920	24	18
SkyDrive App	958,906	0.53	893	16	0
Dropbox App	107,202	0.80	908	24	0
Dropbox Browser	561,242	2.33	2090	69	1096

Table 4.14: Power metrics collected on Nexus 7 during cloud storage scenario using Trepn.

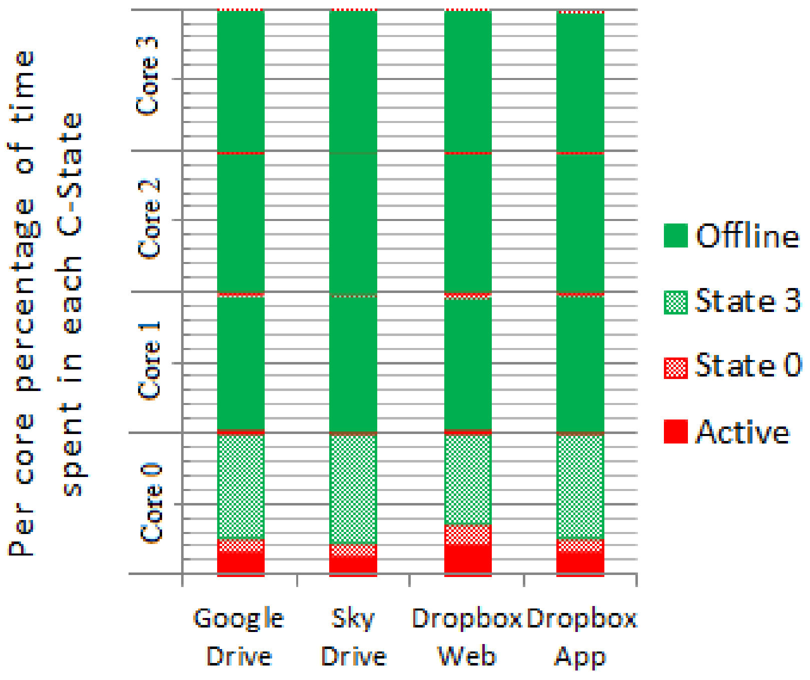


Figure 4.72: Percentage of time spent in each C-State per core collected on Nexus 7 during cloud storage scenario using PowerMon.

Cross-Platform Comparison for Cloud Storage Scenario

In order to compare the energy efficiency of cloud storage apps cross platforms, we provide Table 4.15. The first observation is that SkyDrive ranked number one most energy-efficient app on Surface 2 Pro and iPad Air, however, it ranked the least energy

Application	Surface 2 Pro	iPad Air	Nexus 7
Google Drive	N/A	Rank 2	Rank 3
Google Drive (browser)	Rank 3	Rank 3	N/A
SkyDrive	Rank 1	Rank 1	Rank 4
Dropbox	Rank 2	Rank 5	Rank 1
Dropbox (browser)	Rank 4	Rank 4	Rank 2

Table 4.15: Cross-platform cloud storage energy efficiency ranking.

efficient on Nexus 7. The second observation is that Dropbox ranked number one on Nexus 7, however, it ranked the least energy efficient on iPad Air.

4.5.7 Social Networking Scenario

During social networking scenario, we started the profiling tool along with the 3-minute timer. Next, we launched the social networking app and kept the screen on. Upon the expiration of the timer, we stopped the collection and saved the results. In the case of accessing the social networking app via a browser, we first launched the browser and typed the user name and password. Then, we started the profiling tool along with the 3-minutes timer. Next, we launched the browser and clicked on sign in. We kept the screen on until the timer expired. Next, we stopped the collection and saved the results.

Surface 2 Pro Social Networking Results

We profiled the energy efficiency of the following: Facebook browser, Facebook metro, LinkedIn browser, and LinkedIn metro. Based on the energy consumption values as shown in Figure 4.73, we can rank the energy efficiency of social networking apps as follows: LinkedIn metro, Facebook metro, LinkedIn browser, and Facebook browser. In order to examine the cause of the differences in energy consumption, we examined core idle sleep states, package idle sleep states, total hit count and busy duration, average wake-ups of package and core, and timer resolution as shown in Figures 4.74, 4.75, 4.76, 4.77, 4.78, and 4.79, respectively.

Facebook metro versus Facebook browser: We first noticed that metro version consumed much less platform energy compared to accessing Facebook via a browser, but it consumed more package and core. After further examination, we noticed that the browser version was slightly less active than the metro version and more importantly, the CPU frequency was 26.25% and 66.49% in 2300 and 800 MHz, respectively in the case of metro, whereas, the frequency was 5.14% and 89.33% in 2300 and 800 MHz, respectively, in the case of browser. This can explain why package and core consumed more energy. In order to explain why platform energy consumption contradicted with package and core, we looked at the timer resolution. The browser test case changed the timer resolution to 1 ms, whereas the metro app kept it at 15.6 ms. This change can be directly observed in the average number of wake-ups for both test cases where the browser case had a much larger average wake-ups per second compared to metro. Another direct result from changing the timer resolution is the fact that Facebook (browser) had much more frequent updates from the site as opposed to Facebook (metro). These updates resulted in keeping the Wi-Fi radio active for longer duration, consuming more energy, thus increasing the platform energy consumption (even though package and core consumed less energy when comparing the alternative case of metro.)

Facebook browser versus LinkedIn browser: Facebook browser consumed more platform energy than LinkedIn browser even though total busy duration and average wake-up per package and core are greater in the case of LinkedIn compared to Facebook. We noticed that if we add up the total percentage of active time for core 0 and core 1 for Facebook and LinkedIn, we get the exact same value. However, in the case of Facebook, core 0 was active for longer duration than core 1, unlike LinkedIn, which had approximately the same percentage of active duration for both cores. As a result, percentage of package active duration for LinkedIn was lower than Facebook, which means that both cores were approximately active at the same time, thus they allowed the package to go to sleep faster, resulting in energy savings.

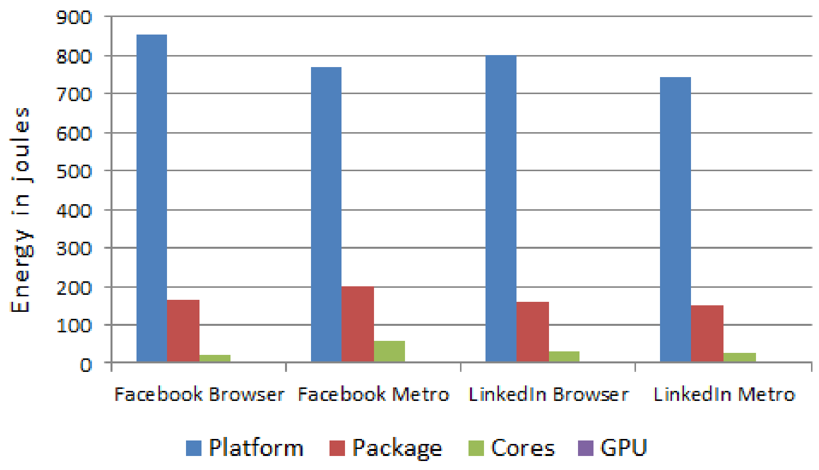


Figure 4.73: Energy consumed by Surface 2 Pro during social networking scenario.

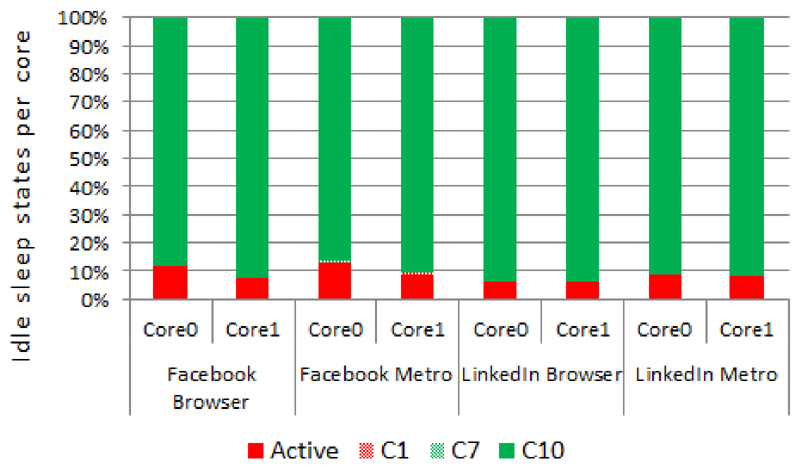


Figure 4.74: Idle sleep states percentage per core collected on Surface 2 Pro during social networking scenario.

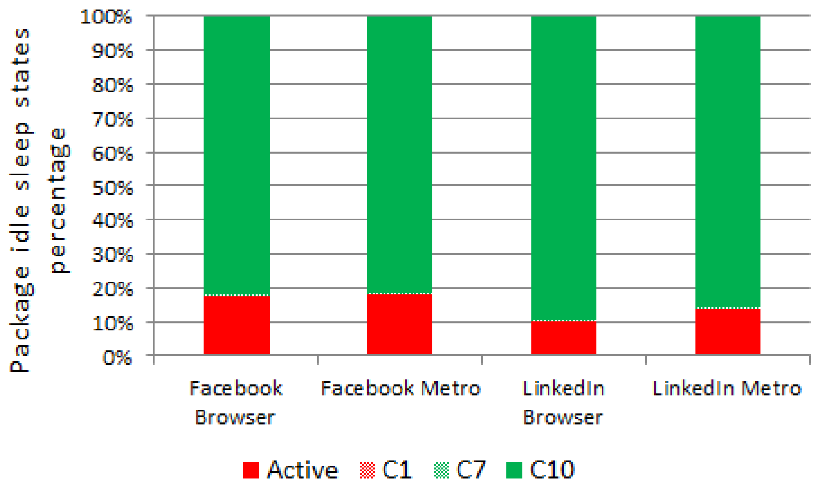


Figure 4.75: Package idle sleep States percentage collected on Surface 2 Pro during social networking scenario.

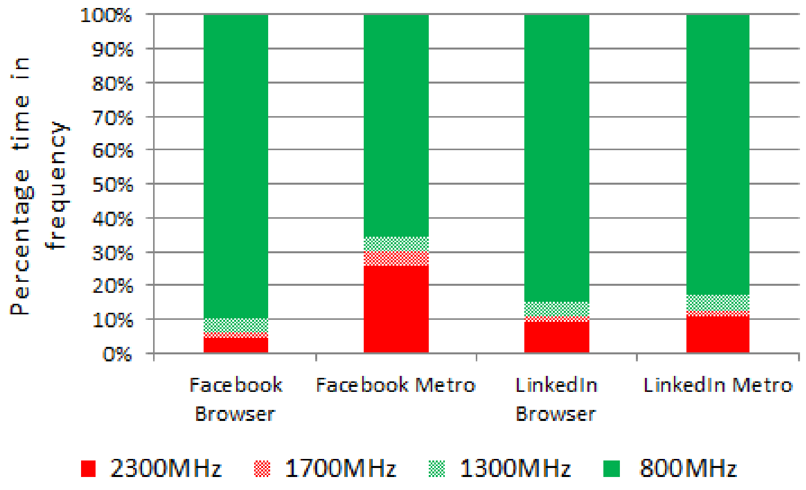


Figure 4.76: Core frequency distribution collected on Surface 2 Pro during social networking scenario.

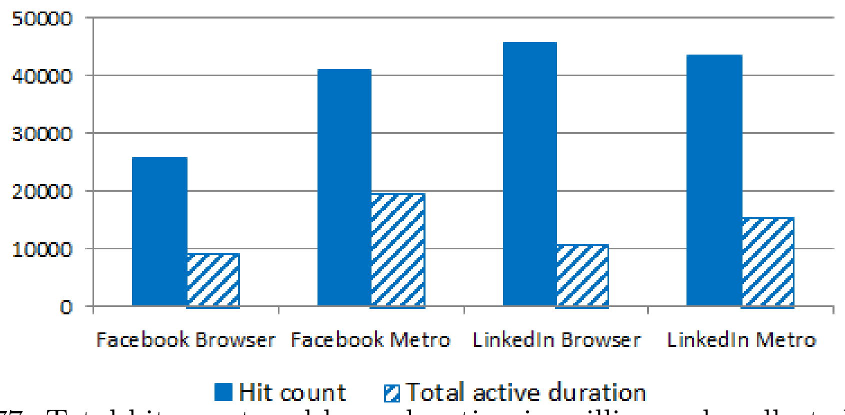


Figure 4.77: Total hit count and busy duration in milliseconds collected on Surface 2 Pro during social networking scenario.

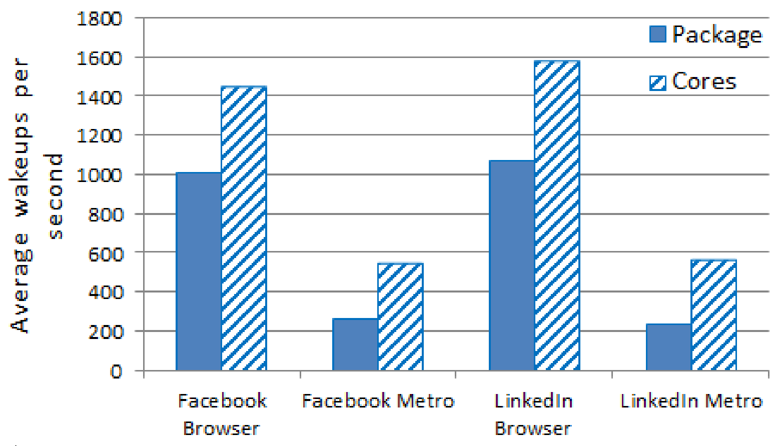


Figure 4.78: Average package and core wake-ups per seconds collected on Surface 2 Pro during social networking scenario.

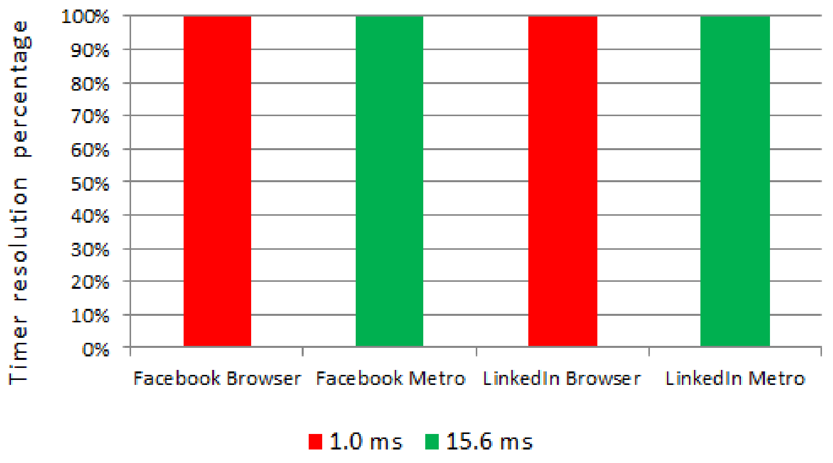


Figure 4.79: Percentage of time spent in each timer resolution.

iPad Air Social Networking Results

We profiled the energy efficiency of the following social networking apps: LinkedIn browser, LinkedIn app, Facebook browser, and Facebook app. Figure 4.80 displays the variation of energy consumption levels during the entire test duration. The average energy level is: 1.82, 3.96, 4.21, and 5.1 for LinkedIn browser, LinkedIn app, Facebook app, and Facebook browser, respectively. As a result, we can rank the energy efficiency of social networking apps as follows: LinkedIn browser, LinkedIn app, Facebook app, and Facebook browser. In order to examine the differences in energy consumption, we examined the total CPU activity percentages and graphics activity percentages as shown in Figures 4.81 and 4.82. The average CPU activities are 5.66%, 4.40%, 5.51%, and 8.08% for LinkedIn browser, LinkedIn app, Facebook app, and Facebook browser, respectively. In addition, the average graphics activities are 2.88%, 2.57%, 4.17%, and 5.01%, respectively, for the same order of applications as the average CPU activities.

LinkedIn browser versus LinkedIn app: LinkedIn browser was more energy efficient than the app version despite the fact that both average CPU utilization and average graphics utilization were greater in the case of browser compared to the app version. As a result, we examined the network activities for both cases. The main differences with network activities is that the browser version received 60 bytes at 3-seconds interval during a large portion of the test (in addition to the large packet sizes at the beginning of the test and at around 40 seconds intervals); on the other hand, the app version received 60 bytes at alternating 2-seconds intervals and 1-second intervals (in addition to the large packet sizes at the beginning of the test and at around 40 seconds intervals). That means using the browser, the Wi-Fi radio was enabled to enter deep sleep states for longer duration than the app version, thus making the LinkedIn browser version more energy efficient than the app version.

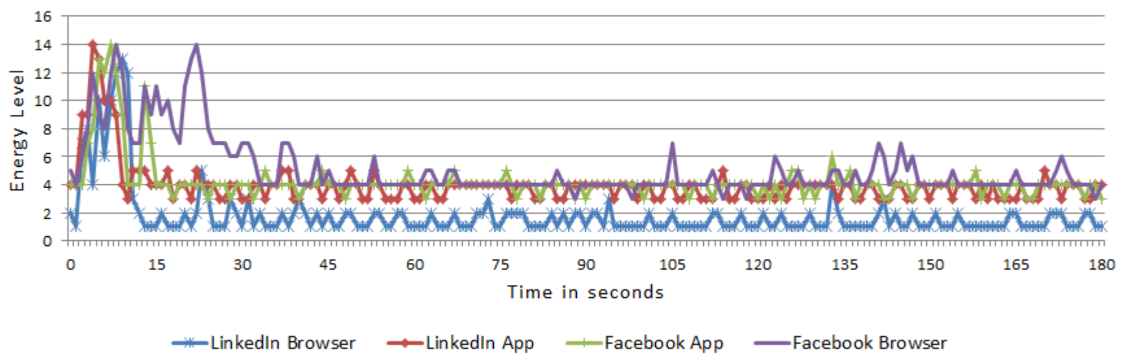


Figure 4.80: Energy level collected on iPad Air 2 during social networking scenario using Instrument.

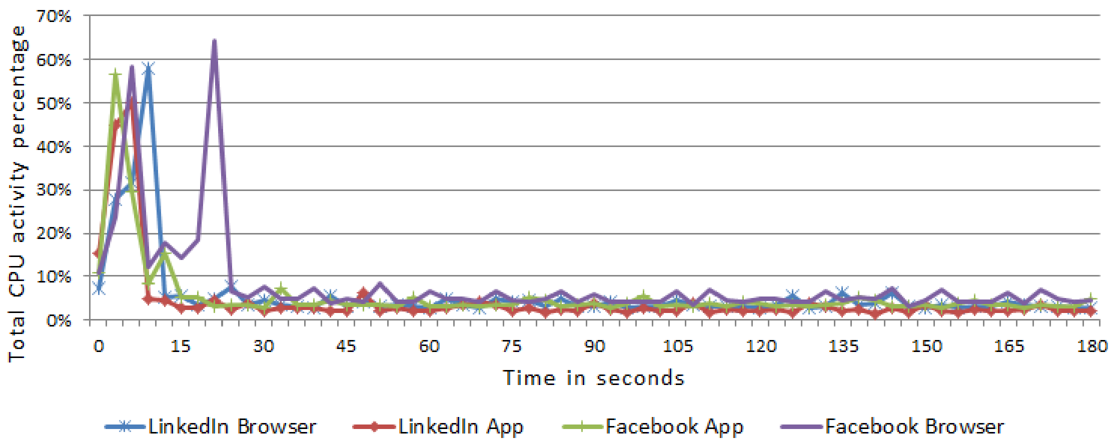


Figure 4.81: Total CPU activity percentage collected on iPad Air 2 during social networking scenario using Instrument.

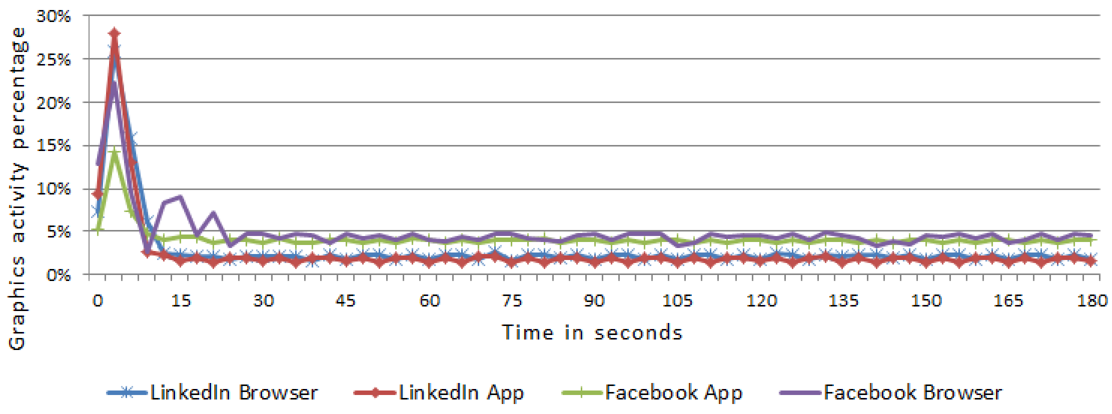


Figure 4.82: Graphics activity percentage collected on iPad Air 2 during social networking scenario using Instrument.

Nexus 7 Social Networking Results

We profiled the energy efficiency of the following cases: Facebook app, Facebook accessed through a browser, LinkedIn app, and LinkedIn accessed through a browser. Table 4.16 represents the power metrics collected using Trepn and Figures 4.84 and 4.83 represent the percentage of C-States and frequency. Based on the results, we can classify the energy efficiency for social networking apps: LinkedIn app, LinkedIn browser, Facebook browser, and Facebook app.

LinkedIn app versus all: LinkedIn app was the most energy efficient social networking app due to the fact that it had the least amount CPU usage percentage which resulted in the lowest active core c-state and the lowest frequency compared to all other apps and browsers versions.

LinkedIn browser versus LinkedIn app: We noticed similar observation to previous scenarios in terms of average virtual memory utilization when comparing browsers and app versions of an application, with the exception that even CPU percentage utilization of the browser version was greater than the app version. This fact also lead to higher percentage of active C-states and frequency. Thus, the explanation of why LinkedIn app version consumed less average power compared to LinkedIn browser.

Facebook App versus Facebook browser: Unlike previous observations where the app version consumes less average power than the browser version, Facebook app consumed approximately 25.52% more power than the browser version. We noticed similar observation to previous scenarios in terms of percentage of CPU utilization where the app version had higher CPU utilization than browser; however, in this case, the average virtual memory utilization of Facebook app was also relatively high and close to the browser version. In addition, Facebook app had a sophisticated user interface. As a result, we checked the GPU load percentage and noticed that the app version utilized the GPU right after launching the app, 17% more than the browser version did upon signing in.

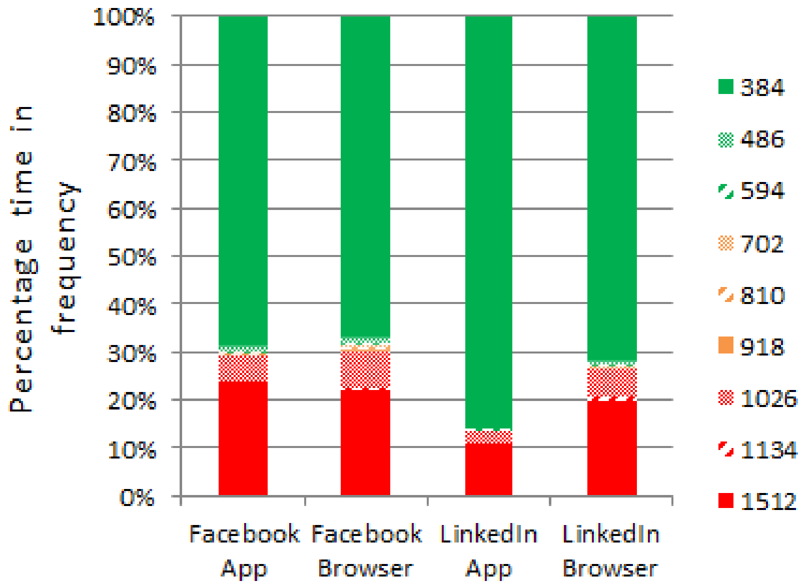


Figure 4.83: Percentage of time spent in each frequency collected on Nexus 7 during social networking scenario using PowerMon.

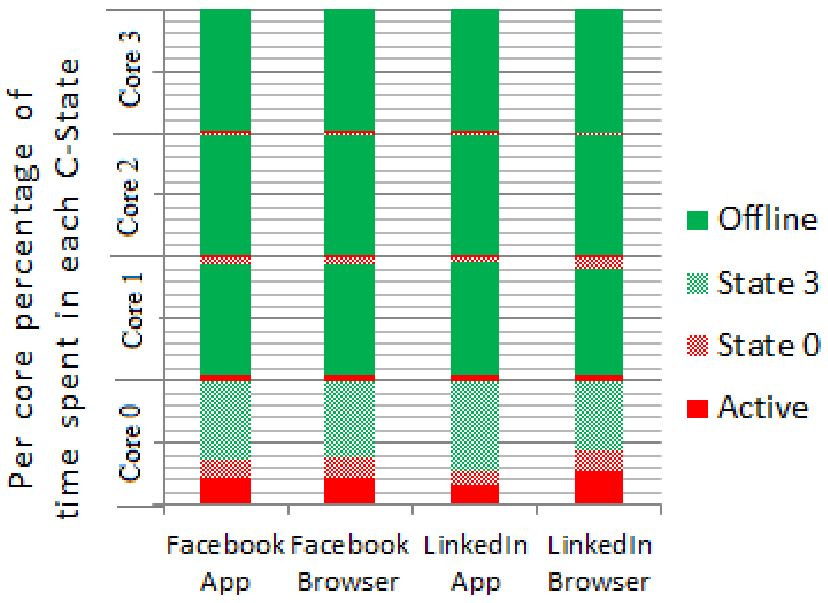


Figure 4.84: Percentage of time spent in each C-State per core collected on Nexus 7 during social networking scenario using PowerMon.

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
Facebook App	1,211,526	1.74	1817.81	49	4
Facebook Browser	965,208	1.35	2089.94	72	1023
LinkedIn App	426,946	0.04	946.38	31	0
LinkedIn Browser	640,353	0.55	2131.36	70	1024

Table 4.16: Power metrics collected on Nexus 7 during social networking scenario using Trepn.

Application	Surface Pro 2	iPad Air	Nexus 7
LinkedIn	Rank 1	Rank 2	Rank 1
LinkedIn (browser)	Rank 3	Rank 1	Rank 2
Facebook	Rank 2	Rank 3	Rank 4
Facebook (browser)	Rank 4	Rank 4	Rank 3

Table 4.17: Cross-platform social networking energy efficiency ranking.

Cross-Platform Comparison for Social Networking Scenario

In order to compare the energy efficiency of social networking apps cross platforms, we provide Table 4.17. In general, LinkedIn was more energy efficient on all platforms compared to Facebook. In addition, both LinkedIn and Facebook had the app version less energy efficient than the browser version on one of the platforms.

4.5.8 E-mail Scenario

During the e-mail scenario, we started the profiling tool along with the 3-minutes timer. Next, we launched the e-mail app and kept the screen on. Upon the expiration of the timer, we stopped the collection and saved the results. In the case of accessing the e-mail via a browser, we first launched the browser and typed the user name and password. Then, we started the profiling tool along with the 3-minutes timer. Next, we launched the browser and clicked on sign in. We kept the screen on until the timer expired. Next, we stopped the collection and saved the results.

Surface 2 Pro E-mail Results

We profiled the energy efficiency of the following: Windows Mail metro, GMail metro, and GMail browser. Based on the energy consumption values as shown in Figure 4.85, we can rank the energy efficiency as follows: Windows Mail metro, GMail metro, and GMail browser. In order to examine the cause of the difference in energy consumption, we examined core idle sleep states, package idle sleep states, total hit count and active duration, and average wake-ups per second for package and cores as shown in Figures 4.86, 4.87, 4.88, 4.89, and 4.90, respectively.

Windows Mail versus GMail: We noticed that Windows Mail had the least active percentages of core and package idle sleep states but it remained relatively in high frequency compared to the other two version of GMail. This shows that the processor increased the frequency resulted in faster processing, and thus less active duration of the cores, leading to less energy consumption of the platform. Not to forget that also Windows Mail had less average wakeups per second for both package and cores compared to GMail.

GMail browser versus GMail metro: GMail metro version was more energy efficient than the browser even though it had a higher active core percentage. However, it had a lower active percentage of package. That means that both cores, in the case of the metro version, were active concurrently, which enabled the package to go to the deepest idle sleep state for a longer percentage of time.

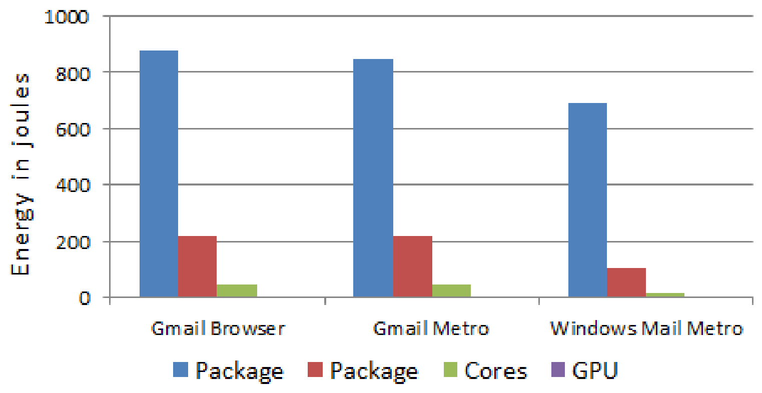


Figure 4.85: Energy consumed by Surface 2 Pro during e-mail scenario.

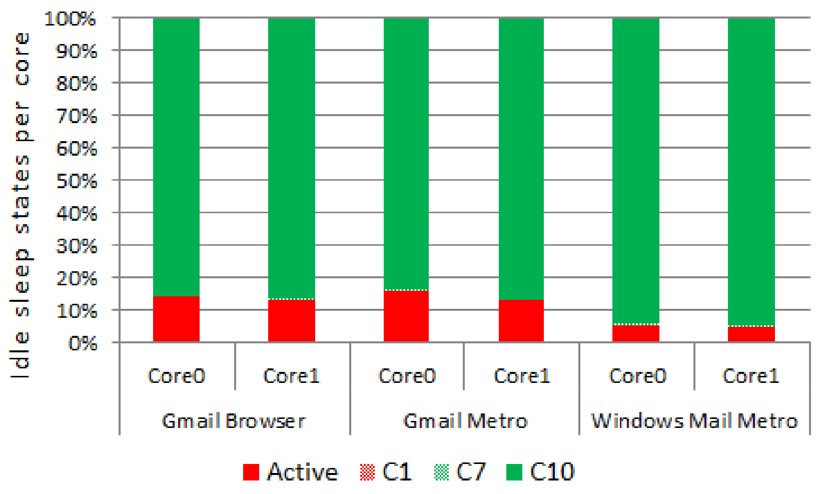


Figure 4.86: Idle sleep states percentage per core collected on Surface 2 Pro during e-mail scenario.

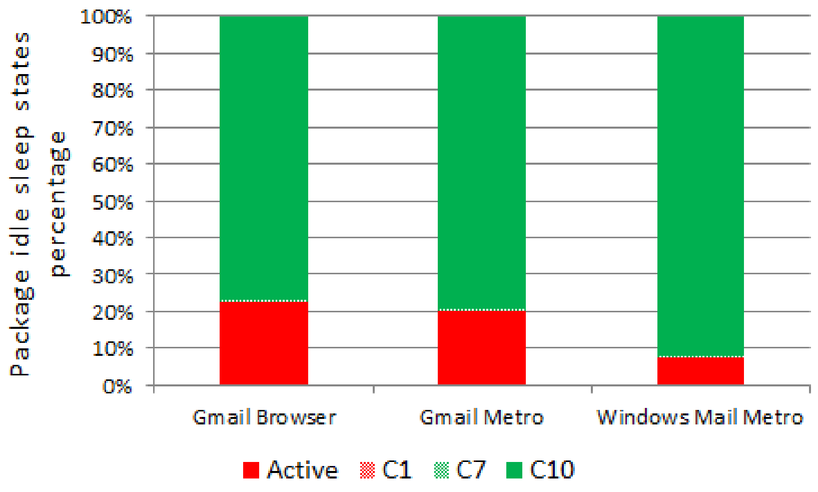


Figure 4.87: Package idle sleep states percentage collected on Surface 2 Pro during e-mail scenario.

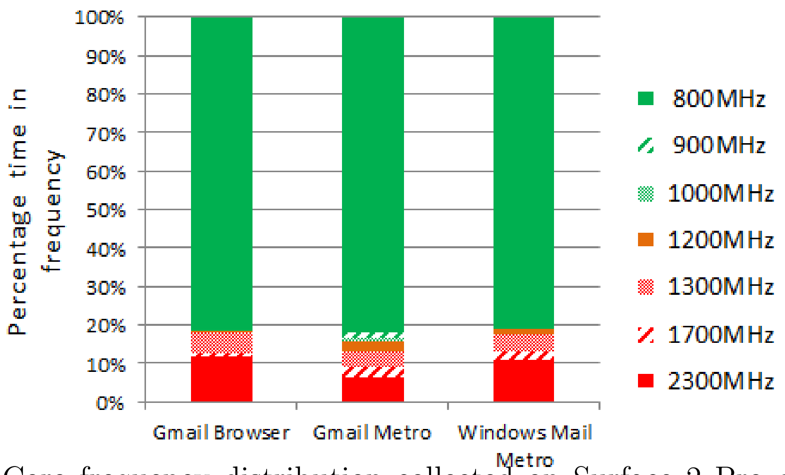


Figure 4.88: Core frequency distribution collected on Surface 2 Pro during e-mail scenario.

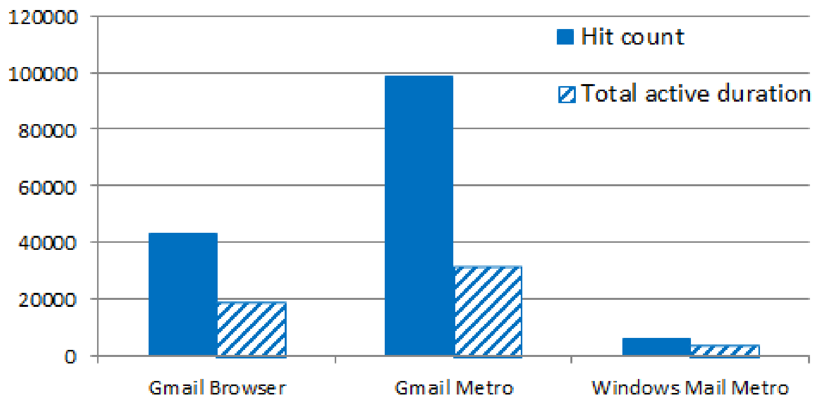


Figure 4.89: Total hit count and busy duration in milliseconds collected on Surface 2 Pro during e-mail scenario.

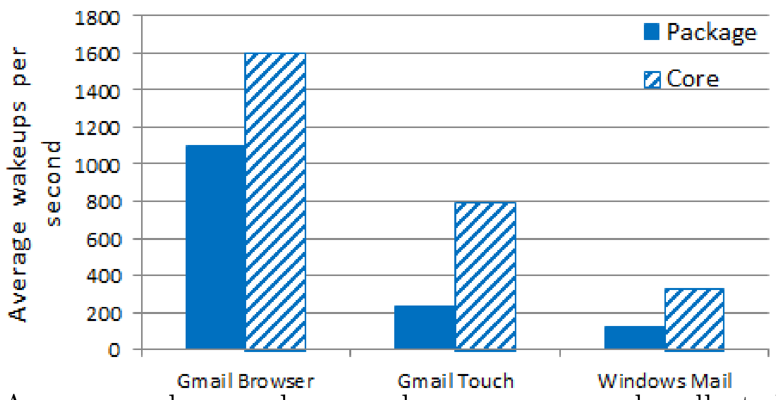


Figure 4.90: Average package and core wake-ups per seconds collected on Surface 2 Pro during e-mail scenario.

iPad Air E-mail Results

We profiled the energy efficiency of the following e-mail clients: Apple Mail, GMail App, and GMail accessed using a browser. Figure 4.91 displays the variation of energy consumption during the entire test duration. The average energy level is: 8.35, 8.42, and 8.59 and the average total CPU utilization is: 3.49%, 5.33%, and 5.46% for Apple Mail, GMail App, and GMail browser respectively. Based on the results, we rank the energy efficiency as follows: Apple Mail, GMail App, and GMail browser. These results are consistent with our observation that utilizing the app version of an application is more energy efficient than accessing it using a browser.

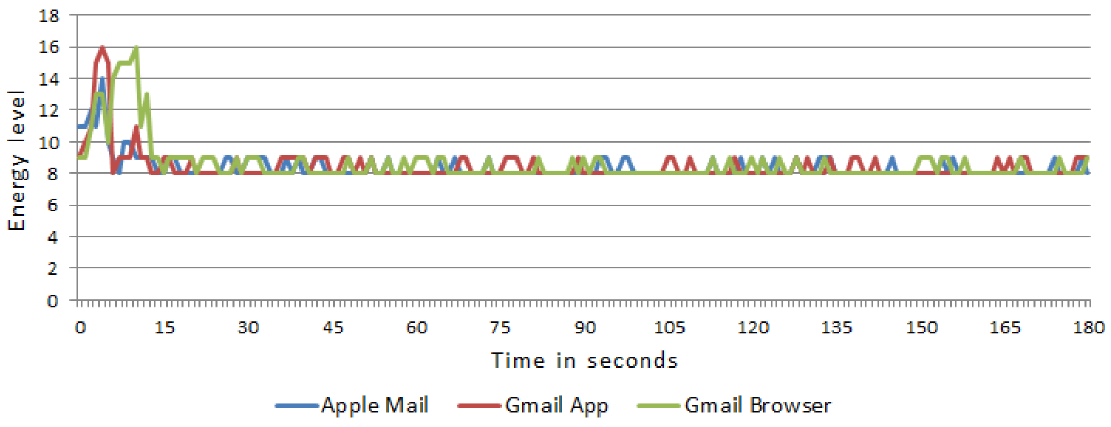


Figure 4.91: Energy level collected on iPad Air 2 during e-mail scenario using Instrument.

Nexus 7 E-mail Results

We profiled the energy efficiency of the following e-mail apps: GMail App, GMail accessed using a browser, and Outlook. Figure 4.92 represents the percentage of CPU utilization over time and Table 4.18 represents the power metrics collected using Trepn. Based on the results, we can rank the energy efficiency of e-mail clients as follows: GMail App, Outlook, and GMail using a browser. Users can actually decrease the energy consumption of the device by 565.91% if they access their e-mail using the GMail app as opposed to accessing GMail through a browser. By examining the core frequency and idle sleep states as shown in Figures 4.93 and 4.94, it is evident that the CPU utilization is directly impacting the percentage spent in high frequency which directly affects the total power consumed by the processor. For example during GMail browser case, the CPU was utilized for an average of 8.28% causing the CPU to remain for 31.94% and 54.35% in 1512 and 1026 MHz, respectively. On the other hand, during GMail app case, the CPU was utilized for an average of 1.69 percent causing the CPU to remain for only 12.78% and 3.12% in 1512 and 1026 MHz, respectively. In addition, the large number of threads and wakelocks when comparing GMail browser versus GMail app caused two cores to be active for large percentages when compared to GMail app where only one core was active for a small portion (12%) and where the remaining three cores were mostly offline.

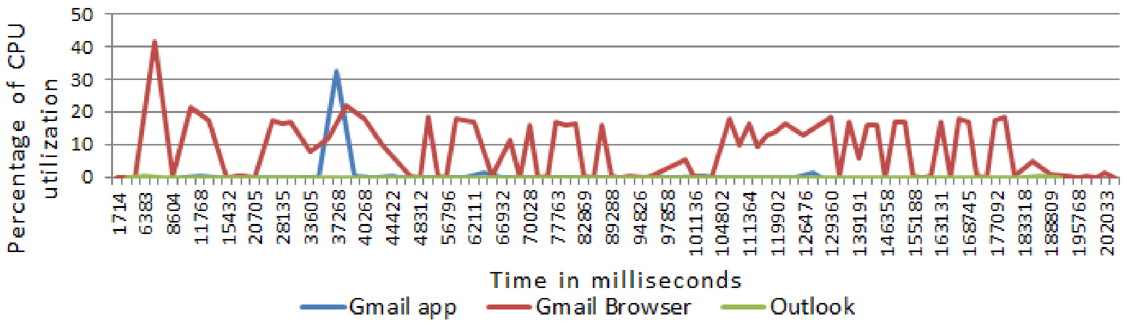


Figure 4.92: Percentage of CPU utilization collected on Nexus 7 during e-mail scenario using Trepn.

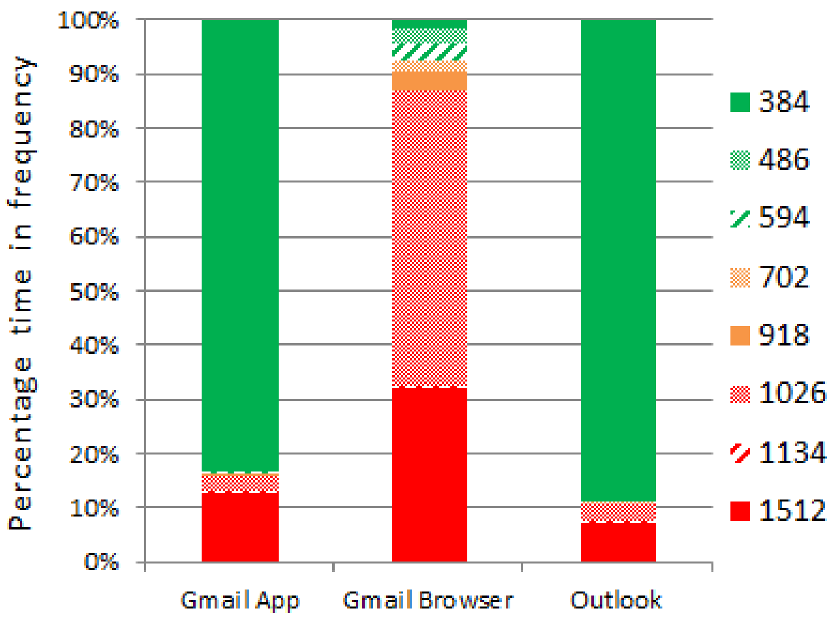


Figure 4.93: Percentage of time spent in each frequency collected on Nexus 7 during e-mail scenario using PowerMon.

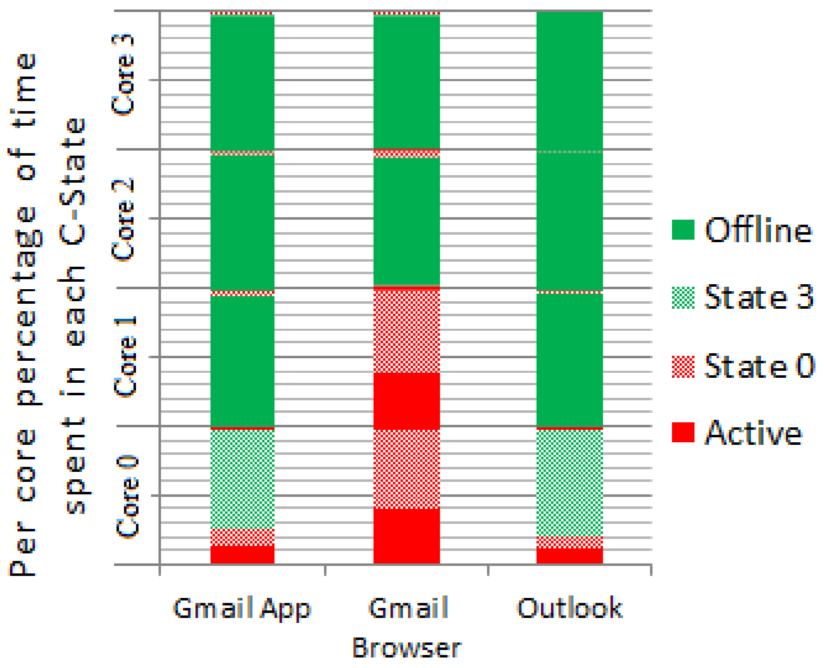


Figure 4.94: Percentage of time spent in each C-State per core collected on Nexus 7 during e-mail scenario using PowerMon.

App Name	Average Power in uW	Average CPU Percentage	Average Virtual Memory	Number of Threads	Total wake-locks
GMail App	207,341	1.69	28	941	34
GMail Browser	1,173,367	8.28	69	2069	911
Outlook	957,468	0.02	40	1767	0

Table 4.18: Power metrics collected on Nexus 7 during e-mail scenario using Trepn.

Application	Surface 2 Pro	iPad Air	Nexus 7
Windows Mail	Rank 1	N/A	N/A
Apple Mail	N/A	Rank 1	N/A
Outlook	N/A	N/A	Rank 2
GMail	Rank 2	Rank 2	Rank 1
GMail (browser)	Rank 3	Rank 3	Rank 3

Table 4.19: Cross-platform e-mail clients energy efficiency ranking.

Cross-Platform Comparison for E-Mail Scenario

In order to compare the energy efficiency of e-mail clients cross platforms, we provide Table 4.19. On all the platforms, e-mail client provided by the OS device company was the most energy efficient. That measn Microsoft's Windows Mail was the most energy efficient on Microsoft's Surface. Likewise, Apple Mail was the most energy efficient on Apple's iPad Air. In addition, for all platforms, accessing GMail via a browser was the least energy-efficient mean sof accessing e-mail.

4.6 Implications

Based on our results, we deduced the following list of implications:

- Having multiple tabs of a single browser open at the same time can have a great impact on the energy consumption of a mobile device. As a result, users should limit the number of open tabs whenever possible in order to increase the battery life of their devices.

- Streaming a video using a native app is more energy efficient than streaming a video using a browser.

- Based on the results collected on Android, apps running within a browser have lower CPU utilization compared to native apps; however, they have more virtual memory utilization, resulting in higher energy consumption than native apps.

- All Apple apps are more energy efficient than all third-party apps belonging to the same app category. Since the energy profiling tool supplied by Apple contained the least precise information compared to other profiling tools, we recommend that Apple should improve their tool in order to provide developers with more variety of metrics to be profiled and with higher precision.

- By comparing applications with the same functionality and running on the same platform, we found that they can vary vastly in terms of energy consumption (more than 50% in some instances). Therefore, due to the lack of point of reference, app developers cannot determine the range of energy efficiency value that they need to target. As a result, there is a need for energy benchmark apps for each category of apps in order for developers to use them as a baseline to compare it with the energy consumption of their apps instead of using device idle energy consumption as the baseline.

- Despite the fact that buffering data can enable Wi-Fi radio to go to deep idle sleep states, it can also increase the power consumption of memory. Since large buffer data are stored in memory and cache, which consume high power, the data size of the buffer needs to be carefully examined because developers need to balance between the energy savings from enabling the Wi-Fi radio to go to idle sleep states and the extra energy consumption due to the increase in memory usage.

- Debugging the energy efficiency is a complicated process where one specific energy metric value in a specific context can mean something completely different in a different context. For example, apps with high wake-up average per second are considered energy inefficient. However, if the processor is active for a large percentage of time and the average number of wake-ups is low, it does not mean that the app is energy efficient. As a result, developers should not focus on one or two energy profiling metrics to profile the efficiency of their apps (e.g, CPU or memory usage). Adequate profiling requires correlation of an extensive set of power metrics and interpreting the data collected in the context of the collection.

- Changing timer resolution on Windows OS and holding wakelocks on Android OS are common practices. They are due to either lack of awareness of the overhead of those energy-inefficient practices on the energy consumption of platforms or developers are making a conscious decision to sacrifice energy efficiency in order to increase performance. Therefore, there needs to be more awareness among developers on the impact of these two metrics on the overall energy efficiency of their apps. Energy efficiency of apps should not be an afterthought but should incorporated in the overall design of the app.

4.7 Related Work

There are several comparative studies for mobile devices. For instance, Gronli *et al.* [48] provided a comparative study where they compared three mobile development environments which are Android, Windows mobile, and Java ME. Qian *et al.* [100] provided comparison and analysis of the three programming models of Android which are Java SDK, C++ NDK, and RenderScript, in order to determine the hands-on programming convenience, runtime behavior, and technical correlation of the different programming models. Moreover, Mullally *et al.* [76] compared performance of enterprise applications on mobile operating systems. They compared two main web services protocols, SOAP and REST on Android and Windows phones. They found advantages and disadvantages based on the metric evaluated.

In order to determine the impact of web apps, Thiagarajan *et al.* [95] focused on energy profiling of web browsing. They provided a tool which can measure the energy needed to render individual web elements such as cascade style sheets (CSS) and JavaScript. They profiled the energy consumption of rendering financial sites, e-commerce, e-mail, and blogging. Then, they made recommendations on how to increase the energy efficiency of web pages. Finally, Gronli *et al.* [49] presented the challenges of testing mobile applications such as on Android and iOS.

Most of the comparative literature focuses on performance because it is impossible to compare, apples to apples, the energy efficiency of an app across platforms. However, since we did not use in our comparison the absolute value of energy, but we used the power consumption behavior, therefore we were enabled to compare the energy efficiency of applications across platforms without violating the comparison rule of similarity.

4.8 Conclusion

In this chapter, we first provided an overview of the top 3 mobile device operating systems (Windows 8, iOS, and Android) followed by a list of top 10 energy-efficient programming rules that developers should consider when developing their apps. Then, we provided a quantitative analysis of eight common app usage scenarios where we compared the energy efficiency of apps of the same scenario on the same platform, followed by cross-platform analysis of the energy-efficiency ranking of app for each scenario. Then, we performed cross-scenarios' energy-efficiency comparison. Based on our case studies, we were able to derive a list of observations and their implications. These implications may be used by developers in order to increase the energy efficiency of their apps.

CHAPTER 5: WHAT IS EATING UP BATTERY LIFE ON MY SMARTPHONE? A CASE STUDY

5.1 Introduction

Smartphones, owned by over 45.5 million people in the United States, are the fastest growing segment of mobile devices [80]. It is forecast that by 2015, smartphone users will increase worldwide to over 1.5 billion, smartphone sales volume will reach 448.8 million, and notebook PCs (Microsoft and Mac) will reach 260 million [24]. Smartphones' increasing popularity stems from their capability to run numerous types of applications, from simple ones, such as playing music, to sophisticated ones, such as group gaming. Despite the faster CPUs and networks, and larger memory, these phones' utility remains limited by their battery life. As a result, the energy efficiency of smartphones is a forefront area of study in the mobile field.

In this chapter, we refer to a smartphone's "idle time" as the case when the screen is off, but applications may be running in the background, even if the smartphone user is not actively using these applications. This idle case is common to many users, who in some cases are warned by smartphone providers about the impact of idle applications. However, users lack understanding of the exact impact and how to reduce it in the event of needing those background applications.

5.1.1 Motivation

Our motivation stems from two facts. The first fact is that several works study the impact of active applications on the battery life during runtime mode, but little attention has been paid to idle time. Energy consumed is a function of the average power consumed over time multiplied by time; therefore, because smartphones remain in this mode for extended periods, such a study is necessary in order to increase the battery life of smartphones without necessarily limiting their key feature of multitasking. The second fact is that smartphones are required to be connected to a network such as 3G or Wi-Fi at all times. Therefore, determining the impact of network connection type is important in order to increase the overall energy efficiency of the smartphone.

5.1.2 Contribution

In this chapter, we make the following contributions:

- We provide detailed case studies of two popular smartphone platforms: an iPhone and an Android. We observed the impact of background applications and network connection type on the devices' CPU utilization and energy consumption. These case studies are particularly important for the following reasons:

    1. Current research literature on power profiling of smartphones focuses on Android because of its open-source nature, while ignoring the number-one competitor, the iPhone. Our detailed iPhone study reduces the research gap.
    2. We provide possible optimization techniques to increase the energy efficiency of smartphones despite the presence of background activities.
    3. We show that even though some concepts are widely known to increase smartphones' energy efficiency (for example, replacing polling functions

with event-driven functions, or using coalescent network activities, especially for not-actively-used systems), the two most popular platforms still have not adopted all these techniques. Therefore, there is still potential for improvement.

4. We aim to increase users' awareness of what is consuming the smartphone's battery life and help them reduce it using the information presented in this chapter.

5.1.3 Organization

The remainder of this chapter is organized as follows. In Section 5.2, we provide models. Then, in Sections 5.3 and 5.4, we provide two case studies using an iOS (iPhone) and Android (Samsung S3), respectively, where we examine the impact of background applications and network connection type on the overall energy-efficiency of the devices. Based on our results, we present in Section 5.5 optimization techniques to increase the energy efficiency of smartphones during idle duration. Finally, we conclude in Section 5.6.

5.2 Smartphone Usage Models

A smartphone's full potential can only be achieved by its ability to connect to the Internet. The common connection models are through 3G cellular data networks and Wi-Fi, in addition to the recent penetration of 4G LTE link which is currently not supported in all areas. The smartphone's usability is diverse and highly dependent on user's demographics. Recent studies by Qian *et.al,* [82] and Falaki *et.al,* [45] revealed that the number of applications used varies from 10 to 90 per user and the number of interactions per day varies from 10 to 200.

Smartphone's usage models are broadly characterized as follows:

- **Streaming media:** This category of applications provides a means to watch, upload, and download videos or music, such as YouTube, Netflix, and Pandora.
- **Social computing:** It refers to applications whose primary role is social interaction, such as blogging, social networking, instant messaging, and e-mailing.
- **Informational:** It refers to applications used to retrieve information, such as news feeds or search engines.
- **Utility computing:** It refers to applications used to perform a computational task or provide a service, such as calculators, calendars, or reminders.
- **Gaming:** It can either refer to local games on the devices or network games where multiple players can play together over the network.

5.3 The Impact of Background Applications and Network Connection Type on iOS Smartphones (iPhone)

This section discusses our iPhone case study, particularly the experimental setup and results.

5.3.1 Experimental Setup and Methodology

During our experiments, we used an iPhone 4S running iOS 5.1.1. We collected our data using Apple's Instrument, which allows dynamic profiling of different performance metrics of the iPhone [1]. We collected the data at 1-second intervals. We performed Test 1 through Test 4 as listed in Table 5.1. Depending on the test type, we started the corresponding applications and let it run in the background. Then, we started the profiler on the phone. Next, we turned the display off. Upon completion of the test period, we stopped profiling. Lastly, we connected the phone to a computer via USB, retrieved the data, and analyzed the results.

5.3.2 Experimental Results

We focused in this chapter on the sleep/wake, energy usage, CPU activity, and network activity profilers of the Energy Diagnostics Instrument.

- **Sleep/Wake:** The sleep/wake status instrument is part of the Energy Diagnostics template. It records the devices' sleep and wake modes. The iPhone has four major modes—sleep, attempting to sleep, running, and waking—ordered from the least power-consuming state to the most power-consuming one. Figures 5.1 and 5.2 represent the results of the sleep/wake modes of Tests 1 through 4 using Wi-Fi and 3G, respectively. States 0, 1, and 2 on the y-axis represent the sleep, attempting to sleep, and running states, respectively. (Waking requires a negligible amount of time, so it is omitted from these graphs.)

- **Energy Usage:** The energy usage instrument does not provide an exact usage value of power; however, it has a scale from zero to 20, ranging from the most efficient to the least. Figures 5.3 and 5.4 represent the iPhone's energy usage using Wi-Fi and 3G, respectively.

- **CPU Activity:** Using the CPU activity instrument, we collected the total percentage of activity over time in addition to each application's activity status change, such as running or suspended. Figures 5.5 and 5.6 represent the percentage of CPU activity using Wi-Fi and 3G, respectively.

- **Network Activity:** Using the network activity instrument, we collected the Wi-Fi bytes in and out, in addition to the cell bytes in and out. Figures 5.7, 5.8, 5.9, and 5.10 represent network bytes in and out using Wi-Fi, using Wi-Fi excluding test 1 (for graph clarity purposes), using 3G, and using 3G excluding test 1 (for graph clarity purposes), respectively.

Table 5.1: Types of Tests

Test number	**Test name**	**Test description**	**Purpose**	**Applications**
Test 1	Background Streaming Apps	Mix of applications with different network utilization with an app streaming in the background	Impact of constantly steaming data in the background	Pandora, Skype, LinkedIn, DropBox, Facebook
Test 2	Network Apps	Mix of applications with different network utilization	Impact of intervallic network connected applications	Skype, Facebook, LinkedIn, DropBox
Test 3	Utility Apps	Mix of utility and non-network applications	Comparing impact on energy consumption of non-network applications compared to network applications	Calculator, PuzzleGame, RollerCoasterGame, UnitConverter
Test 4	True Idle (N/A for Android)	No applications	True idle comparison	None
Test 5	Only Monitoring Apps (N/A for iPhone)	Non-network applications for Android	Get energy consumption baseline	SystemLog, BatteryMonitorPro, SystemPanelPro

Table 5.2: Types of Tests

	Test 1	Test 2	Test 3	Test 4
Average CPU Activity (Wi-Fi)	8.12	1.02	0.38	0.68
Average CPU Activity (3G)	8.86	1.85	0.42	0.35
Average Energy Usage (Wi-Fi)	4.02	1.12	0.36	0.47
Average Energy Usage (3G)	9.85	3.12	0.87	0.86

Table 5.2 summarizes the average CPU activities and the average energy usage for all tests during both Wi-Fi and 3G.

Results of Test 1: During test 1, we noticed the following:

- In the case of Wi-Fi and 3G, the device remained in a running state throughout the test, owing to the continuous streaming of music.
- The energy usage alternated between lower usages to high usages. By comparing the energy usage to the network activities, we noticed that the energy spikes are consistent with the fetching of new network activities.
- When comparing the network activities of Wi-Fi and 3G, it is evident that the quantity of packets are fewer, but larger in size, in the case of Wi-Fi when compared to 3G. This explains why Wi-Fi has a lower number of energy spikes compared to 3G. On average, we can save 59% more energy when using Wi-Fi as opposed to 3G.

Results of Test 2:

During test 2, we noticed the following:

- A similar observation as test 1 regarding the alignment of energy usage to the network activity.
- Unlike test 1, the device alternated between sleep and running states. Also, in the case of 3G, it remained for longer periods attempting to sleep.

- The percentage of CPU usage is lower by 44.86% when comparing Wi-Fi to 3G.
- The average energy usage is lower by 64% when comparing the Wi-Fi to 3G.

Results of Test 3 and Test 4:

During test 3 and test 4, we noticed the following:

- Test 3 and test 4 have similar results, where the device remained in sleep mode for long periods of time. We noticed that the interval of remaining asleep increased over time. We concluded that keeping idle non-network applications does not have a noteworthy effect on the device's energy efficiency.
- We also observed a counterintuitive fact when we compared test 3 and test 4 while Wi-Fi was enabled. Test 4 (which lacks any background applications) had higher CPU activities and higher average energy consumption than test 3 (which has utility applications running in the background). For further investigation, we connected the USB cable between the iPhone and the computer, then ran both test scenarios while collecting the CPU activities via the activity monitor instrument. We noticed the presence of backup function calls, which synchronize the phone with Apple's iCloud when no background applications are present and the phone is in Wi-Fi mode. Apple's iOS is configured to perform automatic backup when the iPhone is connected to power, is in Wi-Fi mode, and is not running any applications. Our original test results of test 4 can be explained by the following: because the phone was connected to Wi-Fi and it was in true idle state, the iOS kept performing poll system calls to check if power was connected in order to perform backup. As a result, CPU activities and energy consumption were higher in test 4 as compared to test 3.

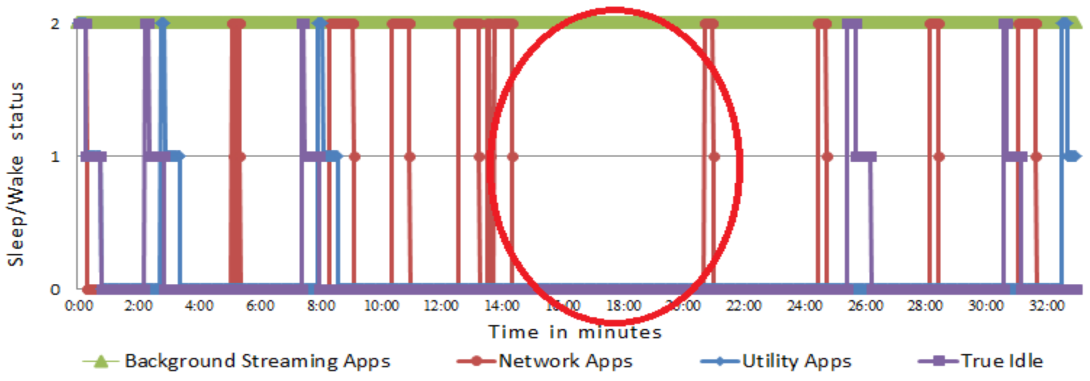


Figure 5.1: Sleep/wake status of iPhone using Wi-Fi.

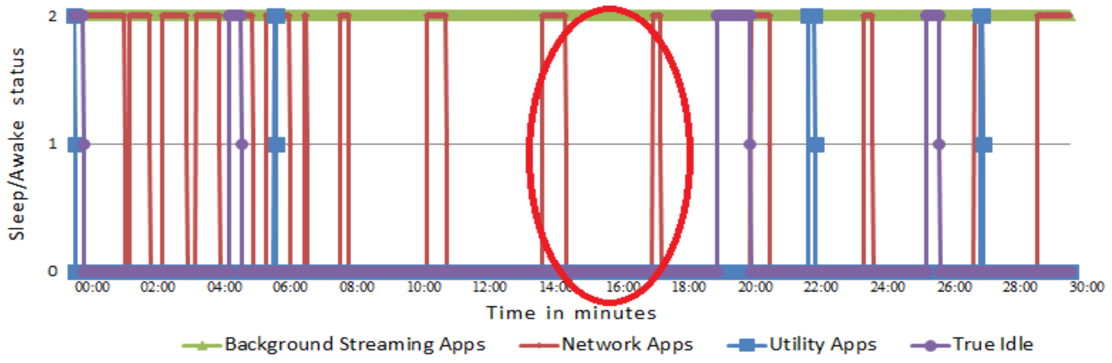


Figure 5.2: Sleep/wake status of iPhone using 3G.

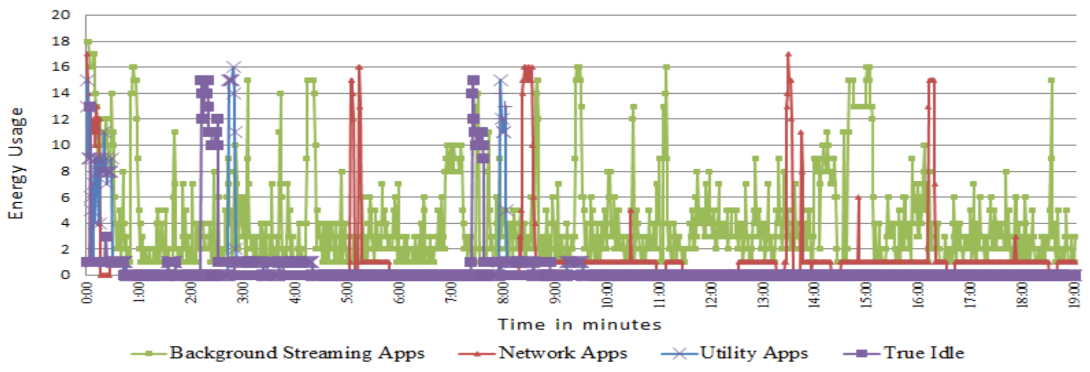


Figure 5.3: Energy usage of iPhone using Wi-Fi.

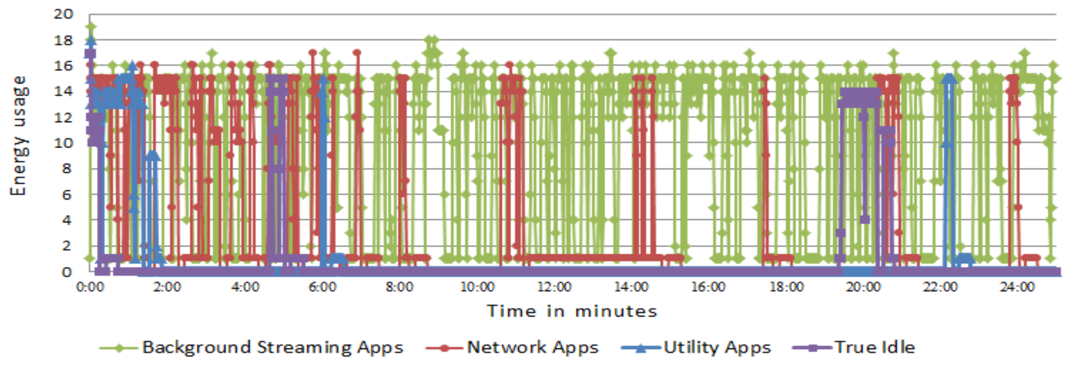


Figure 5.4: Energy usage of iPhone using 3G.

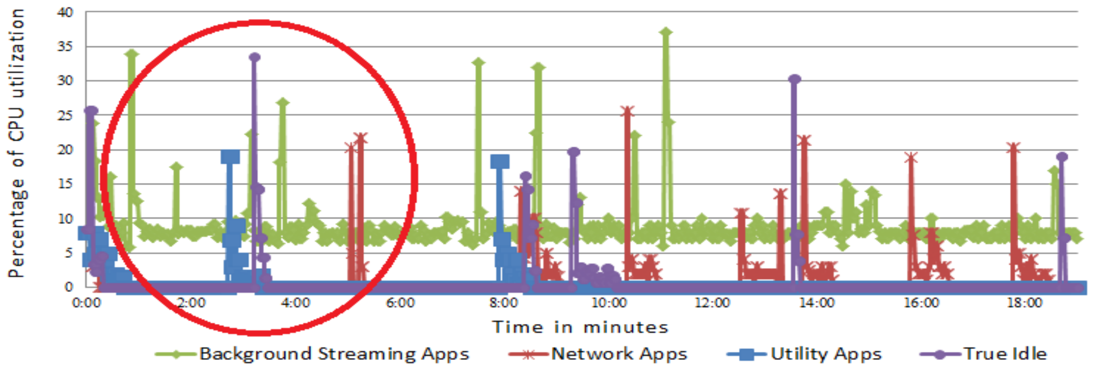


Figure 5.5: Percentage of CPU activity of iPhone using Wi-Fi.

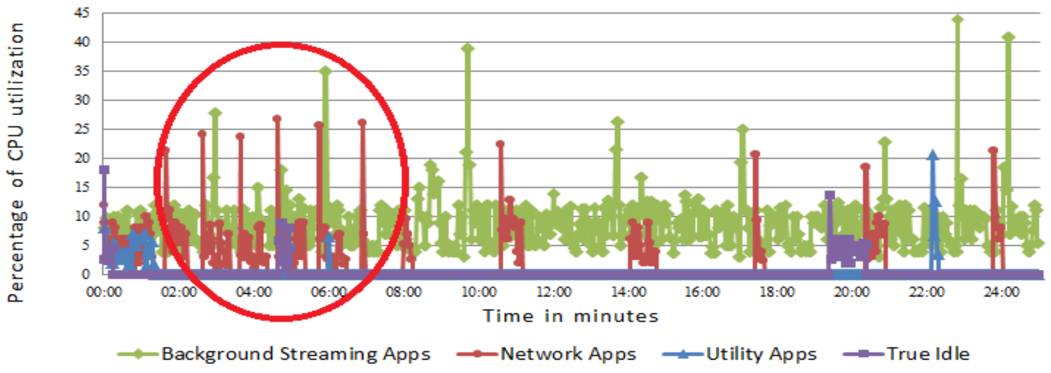


Figure 5.6: Percentage of CPU activity of iPhone using 3G.

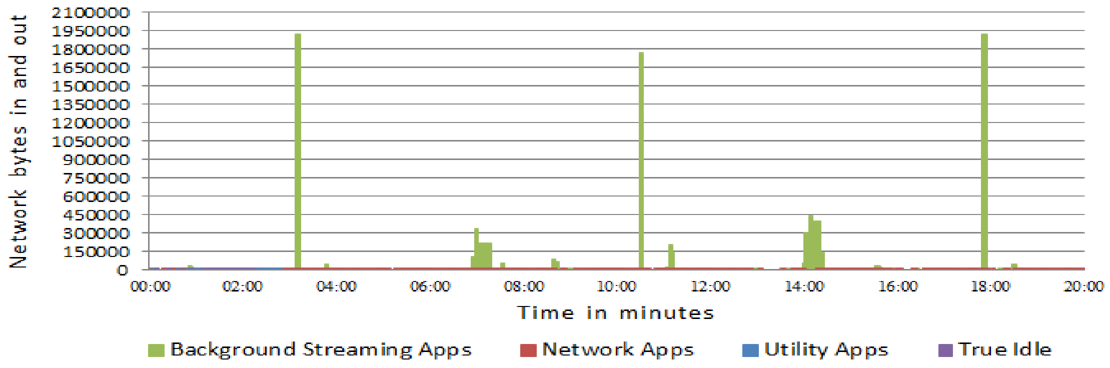


Figure 5.7: Network bytes in and out of iPhone using Wi-Fi.

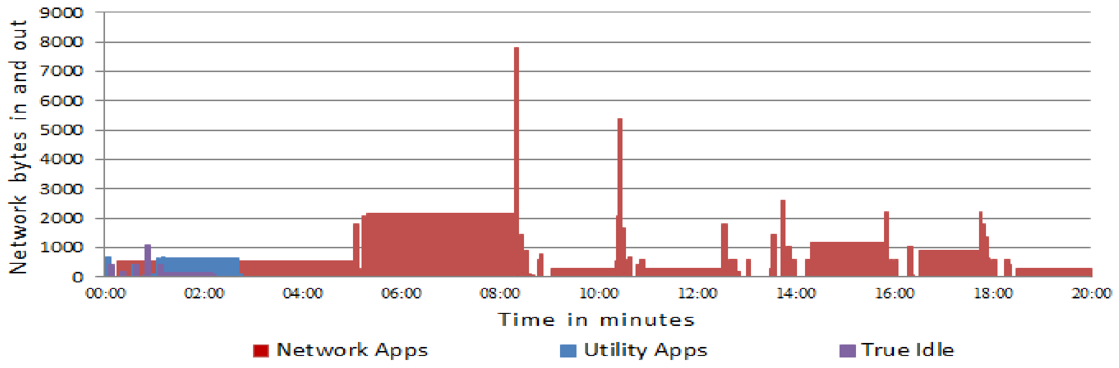


Figure 5.8: Network bytes in and out of iPhone using Wi-Fi excluding background streaming apps test results.

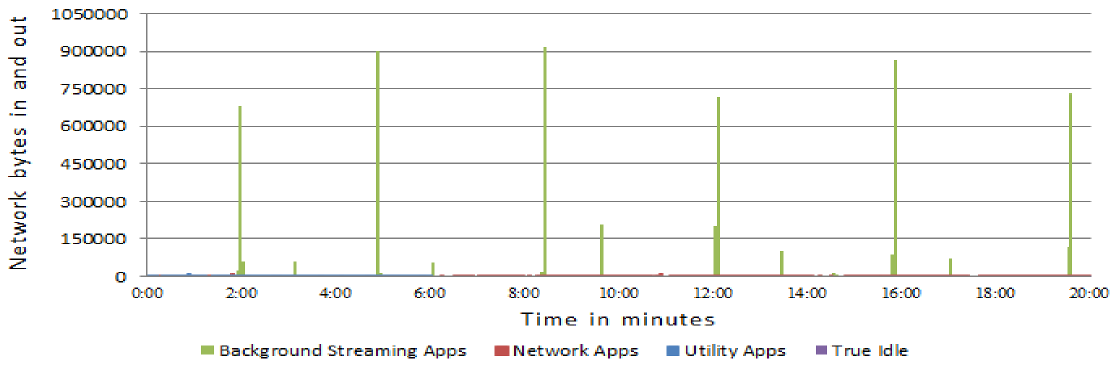


Figure 5.9: Network bytes in and out of iPhone using 3G.

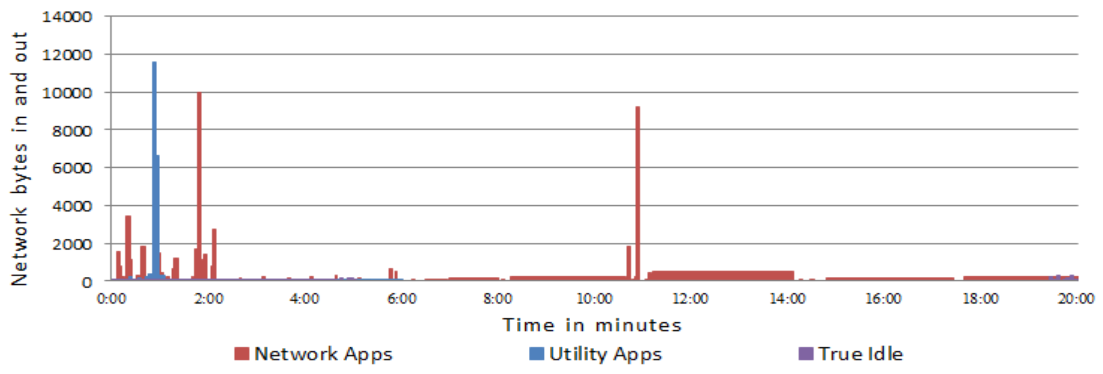


Figure 5.10: Network bytes in and out of iPhone using 3G, excluding background streaming apps test results.

5.4 The Impact of Background Applications and Network Connection Type on Android Smartphones (Samsung S3)

This section discuss our Android case study, particularly the experimental setup and results.

5.4.1 Experimental Setup and Methodology

During our experiments, we used Samsung S3, model number GT-I9300, a rooted Android version 4.0.4, kernel version 3.0.15. We used three applications to collect our data. The first application is Network Log, which collects real-time network activity for each application, including the detailed number of bytes sent by applications and the appropriate timestamp [2]. The second application is Battery Monitor Widget Pro, which records the utilization in mA, the voltage of the battery in mA, and the batterys temperature [3]. The third application is SystemPanel App/Task Manager Pro, which records the system usage, such as CPU usage for each application and overall CPU usage, in addition to other information not used in this paper [4]. We performed tests 1, 2, and 5 as described in Table 5.1.

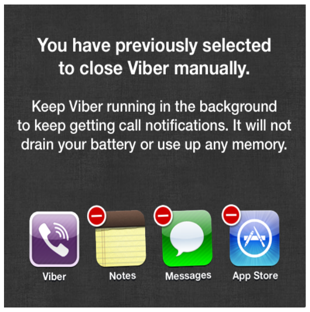
5.4.2 Experimental Results

Using Network Log, we collected the device's network activity. Figures 5.12, 5.13, and 5.14 represent the sum of network bytes in and out of each individual application running on Android while using Wi-Fi for tests 1, 2, and 5, respectively. Similarly, Figure 5.15, 5.16, and 5.17 represent network bytes while using 3G for tests 1, 2, and 5, respectively.

- **Network Usage:**

  It is evident again that Pandora in test 1 dominated the network usage. More network bytes were sent or received periodically during 3G as compared with Wi-Fi. Moreover, Skype and Viber periodically sent or received packets. Note that during test 5, Viber was manually terminated from the task manager and was not supposed to run in the background; however, despite its termination, it remained active. Only the uninstall can prevent its activity.

  During our experiments, we uninstalled Viber from the iPhone and reinstalled it. Upon restart, we received the message, as shown in the Figure 5.11, that Viber will not drain your battery. However, based on our experiments on the Android, we noticed that even when the application is not running in the background, it still periodically sends and receives packets over the network, which can change the radio state from sleeping to running and thus utilize the battery.


You have previously selected to close Viber manually.

Keep Viber running in the background to keep getting call notifications. It will not drain your battery or use up any memory.

Figure 5.11: Viber message notification.

- **CPU Utilization:**

  Figures 5.18 and 5.19 represent the percentage of CPU utilization that we collected from SystemPanel App. The results are similar to the results of the tests on iPhone when comparing the values from Wi-Fi and 3G.

- **Energy Usage:**

  Lastly, we collected the battery consumption from the Battery Monitor Widget. We noticed that, on average, there is a 9% to 14% energy savings when comparing the energy consumption of all the Wi-Fi tests to the 3G tests. There is also 39% to 47% energy savings when comparing the energy consumption between test 5 and test 1.

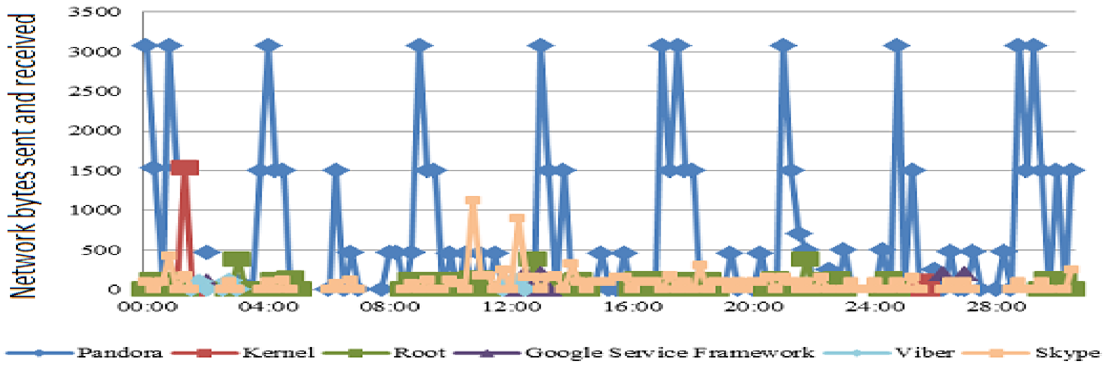


Figure 5.12: Network bytes in and out of Android using Wi-Fi during test 1.

5.5 Optimization Techniques

Based on our observation, we derived a list of optimization techniques to increase the overall energy efficiency:

1. Coalesce of network activities: Every time there is a new network connection, the radio transitions to a full power state and stays in that power state after the transmission is complete [79]. Therefore, it is important to group the network

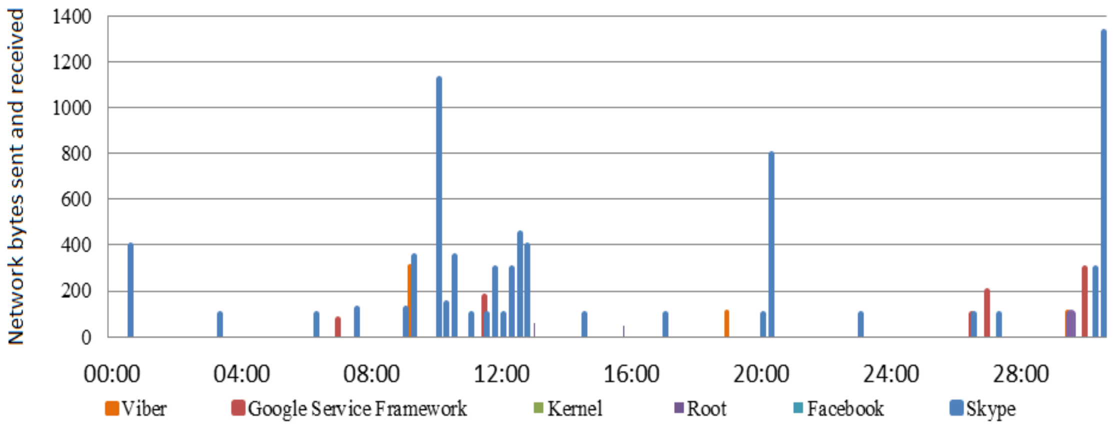


Figure 5.13: Network bytes in and out of Android using Wi-Fi during test 2.

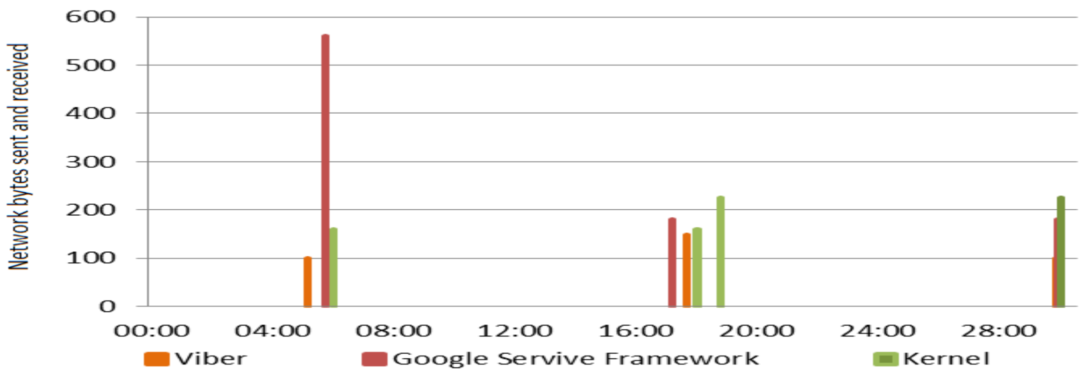


Figure 5.14: Network bytes in and out of Android using Wi-Fi during test 5.

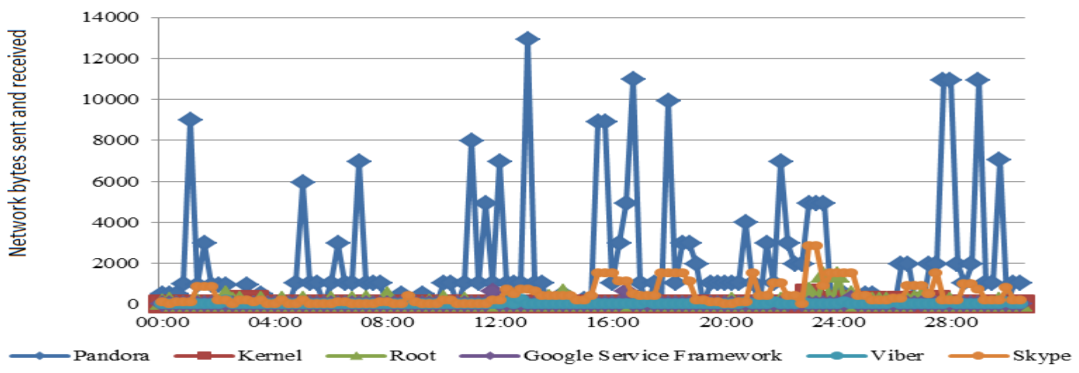


Figure 5.15: Network bytes in and out of Android using 3G during test 1.

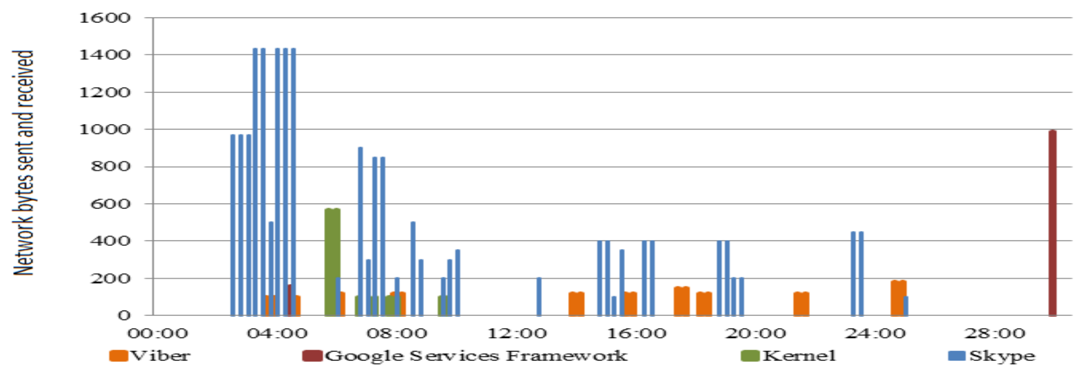


Figure 5.16: Network bytes in and out of Android using 3G during test 2.

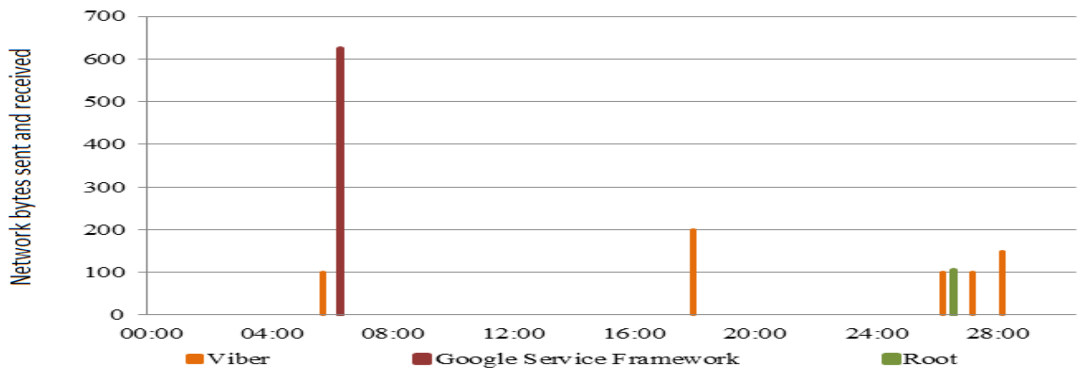


Figure 5.17: Network bytes in and out of Android using 3G during test 5.

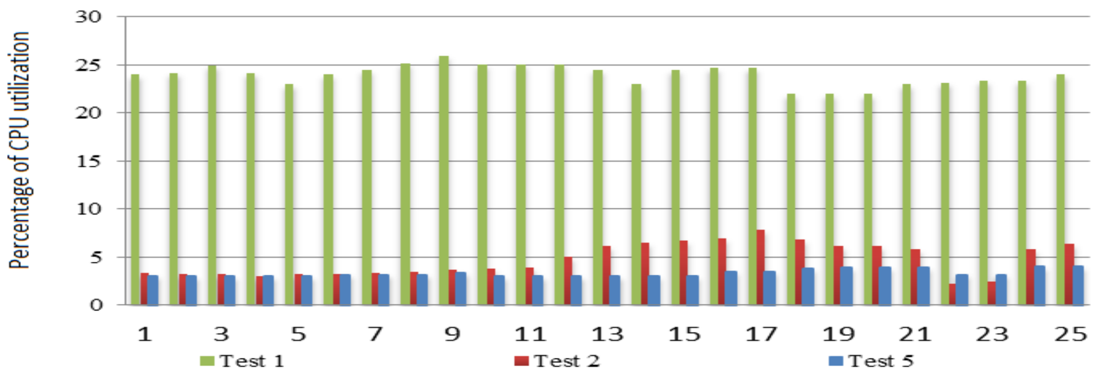


Figure 5.18: Percentage of CPU usage of Android using Wi-Fi.

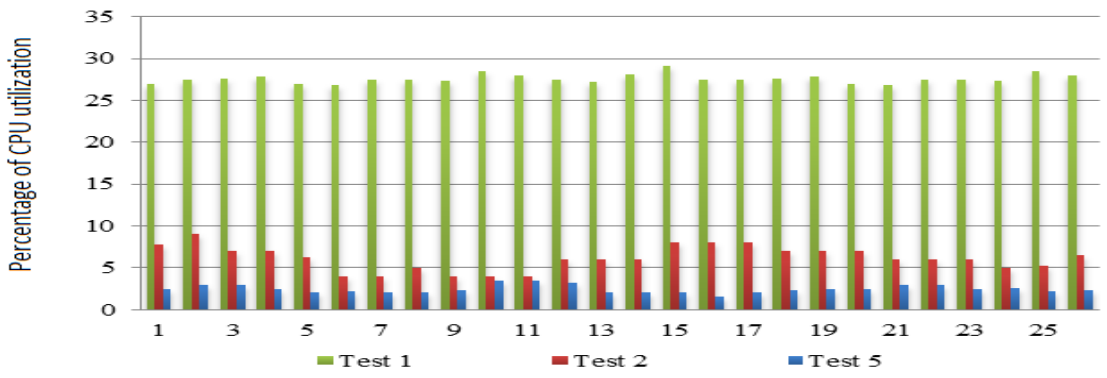


Figure 5.19: Percentage of CPU usage of Android using 3G.

activities as close together as possible, even if they remain in consecutive order, in order to attain longer inactive periods. In particular, we suggest that the network activities performed by the Kernel and Google Service Framework can be coalesced together, because these are system-level activities and should be timed by the OS.

2. Improve the policy of scheduled backup: Backup is a necessary feature given the importance of the information stored on a smartphone. It is a good strategy to perform automatic backups when the following three conditions are met: the phone is in true idle, connected to a Wi-Fi network, and connected to an external power source. However, unlike the current implementation—where the iOS keeps polling to check if the external power source is connected when the phone is connected to Wi-Fi and is in true idle—automatic backup function should be event driven. In other words, if the phone is connected to external power, then the function checks if the other conditions are present to perform automatic backup.

3. Keep the NIC and radio in low-power states: Performance during idle state does not require the same performance requirements as the performance when a user is actively utilizing the smartphone. As a result, a smartphone's efficiency can be further increased by reducing the power states of the NIC and radio during network activity if the I/O is off. Keeping the NIC and radio in low-power

states are not new concepts. However, the current focus is reduction during active mode, but exploring new potential for idle mode is necessary because the requirements during the latter are different from the requirements during active mode.

4. Informed freedom: There are settings in smartphones such as the Android to limit the number of background applications running. However, based on our experiments, not all applications are created equal. For instance, utility applications do not reduce a smartphone's efficiency. Therefore, instead of putting a cap on the number of applications running by automatically forcing them to end, there should be awareness of which applications and network types can reduce the smartphone's efficiency, and which ones do not. Thus, the user can act accordingly.

5. Context Awareness Programming: We demonstrated in this section that when a network application type is running, when Wi-Fi is enabled, it consumes less energy than when it is running on 3G. As a result, software developers should take advantage of this information to create context-aware software that will check the network connection type to determine the tasks the application should perform or, in some cases, ignore until better conditions are present.

5.6 Conclusion

This chapter evaluates the impact of the background application on a smartphone's battery life. We showed on two different platforms, both iPhone and Android, that using Wi-Fi as opposed to 3G will decrease the smartphone's energy consumption and thus make it more energy efficient. We also showed how the network activities (packet size and interval between packets sent and received) directly affect energy consumption and ultimately battery life. Finally, we aim for our findings to be used by smartphone users to extend the battery life of their devices, and for our recommendations for coalescing network activities, improving the policy of scheduled backup, and keeping the NIC and radio in low-power states be adopted by the platform and/or OS providers.

CHAPTER 6: BATTERYEXTENDER: AN ADAPTIVE USER-GUIDED TOOL FOR POWER MANAGEMENT OF MOBILE DEVICES

6.1 Introduction

The battery life of mobile devices is one of their most important resources. However, due to battery size constraints, the amount of energy stored in these devices is limited. As a result, increasing the energy efficiency of these devices is extremely important.

Many factors can impact battery life. Resource utilization by applications running on the platform and the number of powered-on device components the platform has greatly impact battery life. As a result, the platform's power management layer of the device can change the processor frequency or suspend the hard disk in response to utilization. In addition, it can change the device components' power states to an idle sleep state in an attempt to reduce the power consumption of the device components.

There is much research on power profiling of device components or energy profiling of applications in order to enable application developers to debug their applications from an energy-efficiency perspective. However, there is a lack of focus on the end user. What about the user? What if a user needs the platform to last for a specific duration until a particular task is performed, but the battery life is not enough? Can we guide users by giving them options to reach their goal? Will users be willing to completely sacrifice some options in order to achieve their goals? By considering the mobile device as a provider of a collection of resources–similar to a cloud resource provider, which enables users to reconfigure the platform in order to include only the needed resources in order to achieve their goals and completely shut off everything else–then, yes, extending the overall battery life of a mobile device in order to complete a specific task will be possible.

As a result, we developed BatteryExtender, a user-guided tool for power management of mobile devices. The tool's goal is to be strictly software based and to enable users to reconfigure their devices according to the resources needed to accomplish a specific task. BatteryExtender can predict the impact of applications and device components on a platform's overall battery life through minimal energy profiling thus minimizing the power consumption overhead of the tool.

Using BatteryExtender, we were able to reduce the energy consumption of the platform between 10.03 and 20.21 percent, and in rare circumstances, we were able to reduce the energy consumption by up to 72.83 percent. The accuracy rate ranged between 92.37 and 99.72 percent.

6.1.1 Organization

To this extent, we discuss related work in Section 6.2, followed by our motivation in Section 6.3. We present the BatteryExtender design and implementation in Sections 6.4 and 6.5, respectively. We evaluate BatteryExtender in Section 6.6. Finally, we conclude in Section 6.7.

6.2 Related Work

Increasing the battery life of mobile devices has been heavily investigated by researchers. In order to reach this goal, researchers have taken different approaches. We broadly characterize these approaches into the following five categories:

1. Providing power-profiling models of hardware components of mobile devices.
2. Providing estimates of energy consumption of applications, thus enabling application developers to develop energy-efficient applications.
3. Providing APIs for developers to either increase their applications energy efficiency or power profile it.
4. Providing users with power-profiling tools that highlight the impact of running applications on the platform.
5. Using surveys of current power models and/or experiments in order to derive a list of implications and/or recommendations.

Using the first category of research, which provides power-profiling models of mobile devices' hardware components, researchers adopted two techniques: one using external power-measurement tools to accurately measure the power consumption of mobile device components, and one which relied solely on software-based measurement.

Using external power-measurement tools, Carroll and Heiser [32] analyzed the power consumption of smartphone components using a Data Acquisition system (DAQ) with an instrumented platform. They ran various benchmarks in order to accurately measure the power consumption of major components of a smartphone. Based on their analysis, the display, GSM module, graphics accelerator/driver, and backlight were the most power-consuming components. Dong *et al.* [39] also relied on external measurement tools in order to power profile the graphical user interface on OLED displays at the pixel, image, and code levels. They achieved accuracy of 99, 90, and 95 percent, respectively. They built their energy models by measuring the power consumption of the display by collecting the current drawn from a USB interface of a DAQ.

By only relying on software-based techniques for power profiling of device components, Maker *et al.* [68] provided a technique to improve online power modeling in smartphones. They conducted case studies where they profiled power consumption of different smartphone components such as Wi-Fi, GPS, and cellular radio by changing the battery management unit (BMU) sampling rate. As a result, they increased the accuracy of power consumption estimation of those devices. Similarly, Jung *et al.* introduced DevScope [59], an online power-analysis tool for smartphone hardware components, which can accurately build the power models despite the high-interval update rate of the BMU. Sesame [40] is another accurate energy modeling tool that uses a smart battery interface to build accurate power models with low-interval estimation of power consumption. In addition, many software energy-profiling tools utilized Nokia Energy Profiler to build their models. For instance, Perrucci *et al.* [81] conducted a large set of experiments on a Nokia device running Symbian OS 9.2. Their experiments aimed to measure the exact power consumption of all smartphone components while accounting for their different power states. They used Nokia Energy Profiler and verified their results by a multimeter. They determined no significant difference between the reported power consumption values from both. Likewise, Balasubramanian *et al.* [28] used Nokia Energy Profiler to profile network activities of available network technologies. They developed a model for the energy consumption of network devices, which can account for devices' tail power. As a result, they were able to present a method that can reduce the tail power based on the RRC protocol.

Using the second category of research, which provides estimates of energy consumption of applications, a significant number of tools were developed enabling software developers to debug the energy efficiency of their applications. First of all, each operating system provider supplies its developers with tools to power profile their apps, in an effort to highlight the importance of power consumption of apps and enable developers to optimize their apps in terms of power consumption. For instance, Apple provides its iOS and OS X app developers with Instruments [14], a power-profiling tool that enables developers to profile the apps' utilization of resources, such as CPU, Wi-Fi, memory, and energy. Likewise, Microsoft provides an energy consumption tool within the Visual Studio 2013 environment [92], where developers can get power estimation of their apps while prioritizing information on the basis of the metrics under their control. The tool does not require any specialized hardware; however, it doesn't offer accurate power characterization of the device. WattsOn [74] is another tool aimed for application developers. It allows them to focus on the energy efficiency of their code by mimicking the Windows Phone platform and estimating the app's energy consumption on the basis of empirically derived power models made available by either the smartphone manufacturer or mobile OS platform developers. Likewise, Eprof's [79] main goal is to capture and account for the power usage of the program entity by precisely accounting for the entitys effect on components' power state and accounting for the power consumed by the component even after the entity completed its functionality. The tool can be used by application developers in order to find the source code of energy bugs such as "wakelock bugs." In addition, Pathak *et al.* [80] provide fine-grained power modeling for smartphones using system call tracing, which uses two types of models: utilization-based and nonutilization-based power behavior. This technique did not simply enable them to account for components' power based on their state, but also for the components' tail power, and then associate the values with the application responsible for the power consumption. Other tools are aimed at an even-lower level, focusing on enabling architects and developers of compilers such as Wattch [31]. Wattch is a framework that can analyze and optimize microprocessor power dissipation at the architecture level.

The third category of research, which provides APIs for developers, resulted in the following frameworks. Senergy was developed by Kansal *et al.* [61]. It includes an API that can be used by developers of context-aware applications in order to enter latency, accuracy, and battery (LAB) requirements independent of sensors and inference algorithms. Then, Senergy attempts to meet developers' LAB requirements by adapting as the hardware changes. Another framework example is SystemSens [44], developed with the goal of monitoring usage of smartphones' research deployment. It has a client-server model where the apps on smartphones (clients) send periodical information to the server. A subset of the events sent are related to battery usage, screen status (on or off), service logs, and network traffic statistics such as Wi-Fi signal strength, just to name a few. Application developers can use the AIDL interface to be treated as a virtual sensor of the framework and thus collect context and power utilization data related to the application. This information can be collected and monitored by the application developers in order to increase the energy efficiency of their applications.

The forth category of tools is aimed at users in order to highlight the impact of running applications on their platform. Most of the tools in this category rely on collective information to build the energy consumption models. For instance, Carat [78] is a tool that sends coarse-grained statistics to servers residing in the cloud. The statistics sent include battery usage, running apps, the device model, and the operating system. Based on the data collected from the pool of users, the tool can profile the application's impact on battery life and send notifications to users such as the best configuration properties of their specific platform in order to increase battery life while running a specific application. Carat also notifies users about power-hungry apps and apps that contain energy bugs. Likewise, Wang *et al.* [99] used a collaborative approach to estimate the power consumption of mobile applications. They collected data from 120,000 Android users for about four weeks. The information collected contained battery traces and application switching events. Then, they used the data to build their power estimation model for mobile applications.

Lastly, an example of research that relied on surveys and experiments was presented by Shye *et al.* [89], who characterized user activities in smartphones over a period of six months by logging all the activities. Then, they deduced a list of implications for future power-consumption studies for smartphones in addition to providing recommendations for platform and software developers. Similarly, Vallina-Rodiguez *et al.* [97] highlighted the importance of considering the context when evaluating energy demands and resource availability of handheld devices. By collecting a set of data related to OS usage, battery statistics, network profile, and usage, in addition to CPU, screen, and USB, they were able to show that user behavior defines usage patterns and energy consumption. Abdelmotalib and Wu [26] recommended using Wi-Fi instead of Bluetooth for transmitting large data size in order to improve the energy efficiency of mobile devices. They based their recommendation on surveying studies that focused on power consumption of device components of mobile devices. Finally, Datta *et al.* [36] discuss Android power-management techniques, provide a survey of power-saving apps for Android, and derive their limitations. In addition, they suggest a future direction in power management specifically focusing on the client-server model for power profiling and understanding user behavior. They then provide privacy and security concerns when using that approach. In addition, they proposed a photovoltonic cell as an external power source provider that can be integrated into the screen in order to take advantage of the solar power.

Our research differs from the listed related work because we don't simply power profile the devices' components; we also use the information to enable the user to extend the battery life by reconfiguring their device on the basis of our energy-consumption prediction of each component, in addition to the resources needed in order to satisfy the application requirement of resources. We also energy profile an application on the basis of the platform's energy counters and utilization counters.

6.3 Motivation

Our motivation stems from two facts:

- The first fact is the lack of research/tools that enable users to extend battery life on demand. A lot of research focuses on either enabling software developers to increase the energy efficiency of their applications or informing users about power-hungry applications, as shown in the related work section. In addition, we didn't limit our related work research to academic research, but we also surveyed current commercial applications related to battery life that target users. We found an extensive amount of apps (free and paid) on Android devices, such as *Battery, Battery Booster, Battery Defender, Battery Dr. Saver, Battery Extender, Battery Indicator, Battery Info, Battery Mix, Battery Monitor Pro, Top Battery, Easy Battery,* and *One Touch Battery.* Similarly, we found many applications for iOS devices, such as *Battery Doctor, Battery, Battery Expert, Battery Go, Battery Life Pro, Battery Magic Elite, Battery Power, Battery Watch, and Sys Lite.* All these apps displayed the current battery level and either gave an estimate of battery life based on general use, such as *"Audio Playback"* or *"Web Surfing"* duration, or displayed CPU and/or memory usage of apps and enabled users to terminate them. Others showed battery drainage or device temperature over time. However, none enabled the user to precisely extend battery life for a specific time. Even though they show battery duration during the execution of a specific task such as *"Talk Time"* or *"Video Playback time,"* however, their recommendations are general and not specific for a given app. As a result, because apps of the same category (such as *video playback*) can each consume vastly different amounts of energy, the apps' recommendations can be completely off and not useful in many cases. Finally, to the best of our knowledge, we did not find a tool that can answer the question **"what can users do if they need to extend battery life in order to accomplish a**

specific task?"

- The second fact is based on the lack of power-management techniques in response to current and future trends of mobile device evolution. In particular, mobile devices are becoming sophisticated because of the addition of many sensors and device components enabling them to accomplish a variety of goals beyond computation and communication. Some of the goals of this collection of components, sensors, and devices include (but are not limited to) user experience enhancements, health care improvements, environmental monitoring, and tailored advertising. For instance, they can be used to simply enhance the user experience by changing the landscape of the user interface based on the device orientation or changing the display brightness based on the device's surrounding light exposure. Another example is their usability as a means for collaborative diagnostics, such as the case of Carat [78], which collects information from its user base to perform energy diagnostics of applications. Another example is UbiFit Garden [35], which uses mobile sensors to capture physical activities of its users and associate the information with their physical goals. PEIR [77] is another project that uses sensors in mobile devices to alert users about their carbon footprint. Similarly, MIT VTrack [94] is an example of a project that uses mobile sensors to estimate commute time by collecting traffic information from its user base. These examples are only a minuscule subset of techniques used by researchers and industry to take advantage of mobile sensors. Lane *et al.* [66] provide a comprehensive survey of current mobile phone sensing projects and classify them into the following three categories: individual, group, and community sensing. Based on this survey, it is clear that this field of research is gaining traction to become the next hot topic, which can elevate the utilization of mobile devices from "enablers of data access" to "providers of data."

  As a result, we predict that as this field matures, we are going to be introduced to much larger types of sensors enabling the mobile devices to be even more

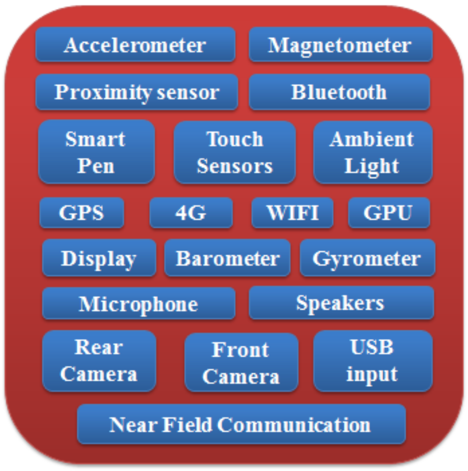


Figure 6.1: Components of mobile devices.

sophisticated. However, as interesting as this new direction seems, the addition of these components can result in an extensive power increase in the platform's overall power consumption, resulting in shorter battery life.

From these two facts, we can deduce a correlation between the cloud concept and mobile trends. More specifically, in a cloud environment, users configure on-demand resources in order to accomplish a specific task. Similarly, if we treat mobile devices as an abstraction of a collection of resources, as shown in Figure 6.1, then we can enable users to reconfigure the device on demand in order to accomplish a specific task. In particular, we consider battery life as a component, and in order to configure a greater amount of battery life, users will need to sacrifice some resources.

6.4 BatteryExtender Design

The BatteryExtender design is based on satisfying the tool's objectives.

6.4.1 BatteryExtender Objectives

Based on our motivation, we clearly define BatteryExtender objectives as follows:

- **Software Only:** External power-measurement tools are expensive and inconvenient for users. In addition, we don't want to feed the tool predefined device component power-consumption values in order to maximize the number of platforms it supports. As a result, we strictly decided to develop the tool using software-only techniques through the utilization of power-consumption metrics provided by the platform.

- **The Tool's Audience:** The tool is not aimed at software developers but everyday users. As a result, it does not require accurate power profiling of platform devices and applications. Its main purpose is to simply enable users to extend the battery life of their mobile devices for a specific duration.

- **Accuracy vs. Overhead:** Continuous power profiling will undoubtedly provide accurate estimations. However, it will also pose some extra overhead. Since the tool is used when the users need to conserve battery life the most, we can sacrifice accuracy in order to reduce power-consumption overhead.

- **On Demand:** We want users to be able to reconfigure their device on demand.

- **User Interactive:** We want the tool to inform users about the impact of platform devices and applications on battery life and enable them to choose the best combination of configurations that to their needs.

This list of objectives is the building block of BatteryExtender's design and implementation.

6.4.2 Design

Based on BatteryExtender's objectives and our energy overhead analysis, we define the tool's architectural design as shown in Figure 6.2. BatteryExtender (BE) consists of the following five components: a calibration module, a user interactive module consisting of user command selection and user interface, an energy profiling module, a power management module, and a monitoring module.

The calibration module should be invoked at least once after the installation of the tool. This module aims to profile the power consumption of platform device components. It requires about 3 hours of profiling time depending on the number of platform components. It requires users to refrain from using the platform during this duration. Upon completion, it produces an XML file with calibration values for all physical components. Users may periodically repeat the invocation of this module in order to increase the tool's accuracy because it is known that a user's behavior over time (such as discharge, charge frequency, and full charge behavior, in addition to the battery wear level) can impact the battery drainage behavior, thus impacting the calibration values.

Using the user interactive module, users can enter the duration by which they want to extend battery life. The selection triggers the energy profiling module. The energy profiling module determines the list of applications running on the platform and calculates the estimate of their energy consumption over the battery life by determining the application resource utilization and energy consumption portion of the application based on the processors' energy consumption. It ranks applications on the basis of their energy consumption and retains the data for the top 5 most energy-consuming applications. This module also determines the current platform settings. It determines each device component state (on or off). Then, it estimates the impact of changing the component state on the platform's battery life based on current battery life duration and using the calibration power estimation data. Upon completion, the energy profiling module updates the user interface of the interactive module with the top 5 power-consuming applications and displays the current platform configuration in addition to displaying the amount of battery life saved/gained by changing the state of each available component.

At this point, the user can check-mark the options to change. Upon option selec-

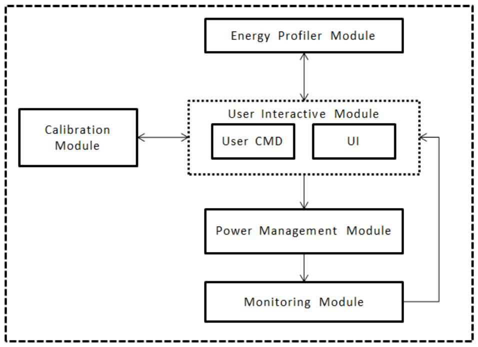


Figure 6.2: BatteryExtender architecture.

tion confirmation, the power management module reconfigures the hardware components in order to satisfy the user's choices and terminate the check-marked applications.

Upon completion, the monitoring module periodically checks the remaining battery life. The goal is to ensure that the remaining battery life satisfies the minimum between battery life extension duration requested by the user and the sum of estimated battery extension duration based on the user's selection. Since the remaining life duration is not accurate, the monitoring module allows few unsatisfactory estimate readings. However, upon reaching a threshold of unsatisfactory remaining battery life duration estimates, the user is notified. Then, the user can either energy profile the platform again in order to reconfigure the platform or accept that the new expected battery life will be the desired remaining battery life.

Finally, in this section we provided a high-level architectural design that satisfies the tool's goals.

6.5 BatteryExtender implementation

BatteryExtender's architectural design can be implemented to any mobile device operating system (OS). However, our target OS is Microsoft platforms starting with Windows 8. The main purpose is because Microsoft, starting with Windows 8, is trying to attract the largest possible market share through appealing to users by providing the same user experience across all its mobile device types, such as laptops and tablets. That means that users get the full system capabilities of a desktop in addition to the tablet experience (similar to Android and iOS) through their Metro Style App model and a full fledged desktop. As a result, a single platform contains an extensive number of components, and our power-management approach can significantly impact the battery life.

Prior to our implementation, we validated our tool's approach through an analysis of current Windows device power-management effectiveness. Then, using experimental analysis, we determined the appropriate collection granularity of battery life in order to increase the accuracy of prediction of battery life extension duration.

During our experiments, we used two different Windows platforms. The first platform is a Dell XPS 12 Ultrabook Convertible, as described in Table 6.1. This platform is a full laptop and can be converted to a tablet as well. In the remainder of this paper, we will refer to this platform as "Dell Convertible." The second platform is the Microsoft Surface 2 Pro Tablet, as described in Table 6.2. In the remainder of this paper, we will refer to this platform as "Surface 2 Pro."

6.5.1 Windows Device Power-Management Analysis

Starting with Windows 8, Microsoft requires that the device components of their platform support five different D-States [10].

The first state is D0, which is the active state where the device consumes maximum power with all its clocks on. The second state is D1, where power consumption and clock are reduced. It is the highest-powered device sleep state, and a clock-gated state where the device preserves it hardware context. The third state is D2, an intermediate device lower-power state. It consumes less power than D1, but most context is lost by the hardware. The fourth state is D3-Hot, which is a very low sleep

Specification	**Description**
Platform	Dell XPS 12 Ultrabook Convertible
Processor	Intel(R) Core(TM) i7-4650U @ 1.70 GHz 2.30 GHz Code Name: HASWELL-ULT
Packages	1
Cores per package	2
Logical processors per core	2
Hard Disk	256 GB Solid State
Memory	8.0 GB
Operating System	Windows 8 Pro
Display	12.5” Full HD (1080p)
Bluetooth	Intel(R) Centrino(R) Wireless Bluetooth(R) + High Speed Virtual Adapter
Wi-Fi	Intel(R) Dual Band Wireless-AC 7260
NFC	NXP NearFieldProximity Provider
Speaker & Microphone	Realtek High Definition Audio
Touch	10 Touch Points
Display Refresh Rate	59 and 60 Hz
Camera	Front WebCam
Sensors	HID Sensor Collection Simple Device Orientation Sensor Microsoft VS Location Simulation Sensor

Table 6.1: Dell Ultrabook Convertible specifications.

Specification	**Description**
Platform	Microsoft Surface 2 Pro
Processor	Intel(R) Core(TM) i5-4200U @ 1.60 GHz 2.30 GHz Code Name: HASWELL-ULT
Packages	1
Cores per package	2
Logical processors per core	2
Hard Disk	64 GB Solid State
Memory	4.0 GB
Operating System	Windows 8.1 Pro
Display	10.6” HD
Display Refresh Rate	59 and 60 Hz
Bluetooth	Marvell AVASTAR Bluetooth Radio Adapter
Wi-Fi	Marvell AVASTAR 350N Wireless Network Controller
Speaker & Microphone	Realtek High Definition Audio
Touch	10 Touch Points Pen
Camera	Microsoft LifeCam Front Microsoft LifeCam Rear
Sensors	HID Sensor Collection Simple Device Orientation Sensor Microsoft VS Location Simulation Sensor

Table 6.2: Microsoft Surface 2 Pro Tablet specifications.

state that consumes less power than all previous states. When a device is in this state, it remains connected to the power source with very low current drawn; however, it can be still detected by the bus. The last state is D3-Cold, the lowest possible sleep state, where the device only receives a trickle of current; it reduces power and clocks to the minimum possible value and only keeps enough to appear on the platform bus and respond to bus commands. After a device enters D3-Cold for a period of time, it gets turned off.

In order to validate our approach of disabling unnecessary components using BatteryExtender, we performed experiments using Surface 2 Pro in order to observe the efficiency of current power management of device components. The goal of this preliminary study is to determine the actual device power sleep states during two scenarios. The first scenario is the platform default state where all components are on, and the second scenario is when we disable the following 10 devices: the Wi-Fi adapter, Bluetooth adapter, HID sensor collection, Visual Studio (VS) location simulation, pen and touchscreen sensors, audio adapter, camera rear and front, and printer queue. We collected the device D-States using Event Tracing for Windows (ETW) for 5 minutes while the platform was in idle with the screen on for both scenarios. We compared the list of devices collected between default settings and reconfigured settings.

By comparing the list of devices, we noted that greater than 50 devices were detected using the default scenario compared to the disabled one. Table 6.3 has the results. The first set of D-States labeled "Idle" represents the actual extra devices that appeared in the report when the platform was in default settings compared to when we disabled 10 devices. The results show that many devices were in the D0 state 100 percent of the time, and just a few were in D0 for 96.9 percent of the time before they entered D2 for the remaining duration of 3.10 percent. As a result, it is evident that many unused devices were in active state with high power consumption, translating into consuming unnecessary battery life.

For further observation, we performed another experiment, where we kept the default settings and ran a local movie for 1 hour, during which time we collected the device D-States. The results are displayed in Table 6.3 in the video playback section. Again, we noticed that many devices remained in active state 100 percent of the time even though the workload did not require it. However, there were some devices that switched to the D2 state for 99.95 percent of the time. In addition, by comparing Idle and Video playback scenarios, we noticed that microphone and speakers were the only devices that were shut off on their own during Idle scenario because they didn't appear during Idle device D-State collection. Another noteworthy observation is that only cameras were in D3-Hot and no devices were in D3-Cold, whereas the majority of devices were in either D0 or D2.

Finally, this experiment demonstrated that a better power-management mechanism was needed. As a result, BatteryExtender can definitely take advantage of this inefficiency in order to enable users to strictly use the needed devices and disable all the rest–thus, increasing the platform's battery life.

Device	Idle					Video Playback				
	D0	D1	D2	D3 Hot	D3 Cold	D0	D1	D2	D3 Hot	D3 Cold
Microsoft Bluetooth Enumerator	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Microsoft Bluetooth LE Enumerator	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
BthLEEnum	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
USB Input Device	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
USB Input Device	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
USB Input Device	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
USB Input Device	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
MMDEVAPI\AudioEndpoint	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
MMDEVAPI\AudioEndpoint	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
mshidumdf	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
MTConfig	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
SensorsServiceDriver	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
UmPass	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
UmPass	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
UmPass	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
USB Composite Device	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
WUDFRd	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Compliant Touch Screen	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%
HID Component	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Component	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Component	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Component	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Component	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Component	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%
HID Component	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%
HID Component	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID Sensor Collection	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID-compliant consumer control device	96.90%	0%	3.10%	0%	0%	0.05%	0%	99.95%	0%	0%
HID-compliant Pen	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%
HP Laser Jet 200 color M251 PCL6 Class	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
HP Laser Jet 200 color M251nw	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
HP Laser Jet 200 color M251nw	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
HP Laser Jet 200 color M251nw	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
HP Laser Jet 200 color M251nw	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Intel HD Graphics Family	4.63%	0%	0%	95.37%	0%	100%	0%	0%	0%	0%
IP Tunnel Device Root	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Lightweight Sensors Root Enumerator	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Mar. AVA. Bluetooth Radio Adapter	98.98%	0%	1.02%	0%	0%	98.98%	0%	1.02%	0%	0%
Mar. AVA. 350N Wireless Net. Controller	100%	0%	0%	0%	0%	98.98%	0%	1.02%	0%	0%
Microsoft LifeCam Rear	0%	0%	0%	100%	N/A	0%	0%	0%	100%	N/A
Microsoft LifeCam Front	0%	0%	0%	100%	N/A	0%	0%	0%	100%	N/A
Microsoft VS Location Simulator Sensor	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Microsoft Wi-Fi Direct Virtual Adapter	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Printer Queue	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Realtek High Definition Audio	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Microphone (Realtek High Definition)	N/A	N/A	N/A	N/A	N/A	99.99%	0%	0%	0.01%	0.%
Speakers (Realtek High Definition)	N/A	N/A	N/A	N/A	N/A	99.99%	0%	0%	0.01%	0%
Surface Cover Audio	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
SWD\PRINTENUM	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
SWD\PRINTENUM	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
SWD\PRINTENUM	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Teredo Tunneling Pseudo-Interface	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%

Table 6.3: Extra device D-States during idle and video playback on Windows Surface 2 Pro compared to the scenario where we disable 10 devices. We highlight in green the devices that switched from active to low device power state when comparing idle to video playback and we highlight in red the devices that should have switched from active to low device power state when comparing idle to video playback due to the long inactive duration.

6.5.2 Analysis of Collection Granularity of Battery Life

Battery life is a major characteristic to determine users' experience of mobile devices. Users want the longest possible battery life. Two variables determine battery life: the battery's power supply and the drainage rate. First, the maximum power supply stored by the battery is directly related to the battery size. The larger the battery, the greater its power supply. However, mobile device users require a lighter system as well, resulting in smaller batteries, leading to limited power supply. As a result, Lithium ion batteries are the batteries of choice for mobile devices because they can store higher energy values per weight unit compared to other types of batteries. Each battery may have a varying capacity based on the amount of energy it can hold. During discharge, the current, which is carried by Lithium ions, moves from negative to positive electrodes, resulting in voltage changes at varying rates based on the amount of power drawn. The amount of power drawn depends on the platform's available components and load. Needless to say, the fewer powered-on components and the lighter the load results in longer battery life because of the decreased power drawn.

The best and most accurate way to determine a platform's power consumption is by using hardware metering equipment. However, since our goal is to strictly use software techniques for our tool, we must translate how the power consumption (discharge rate) translates to the changes in battery capacity and remaining battery life.

Windows platforms support several APIs related to batteries such as (but not limited to) *SYSTEM_POWER_STATE, SYSTEM_POWER_STATUS, SYSTEM_POWER_LEVEL, BATTERY_QUERY_INFORMATION*, *win32_battery*, *CIM_Battery*, and *BATTERY_STATES.* We are interested in collecting the following battery metrics:

- **Battery capacity** reported in milliwatts per hour (mW/h) and denoted as B_{Cap}. It is the amount of energy stored in the battery. The formula to convert mWh to joules is presented in Equation 6.1.

- **Rate** reported in milliwatts (mW) and denoted as $\mathrm{B_{Rate}}$. It is the amount of power drawn from the battery.
- **Battery life remaining**, denoted as $\mathrm{B_{Life}}$. It can be calculated as shown in Equation 6.2.

$$1000mWh \equiv 3.6joule \tag{6.1}$$

$$B_{Life} = \frac{B_{Cap}}{B_{Rate}} \tag{6.2}$$

BatteryExtender uses two different APIs. The first is *GetSystemPowerStatus*, which returns a *SYSTEM_POWER_STATUS* structure that contains *BatteryLifePercent* and *BatteryLifeTime*, among others. The second API is getting a handler to the device interface of the battery in order to collect *BATTERY_QUERY_INFORMATION*, which contains all of the battery information, such as capacity, voltage, rate, and even *BatteryLifeTime*. We used both APIs for the following reason: the update rate of *BatteryLifeTime* of each API is different. The first API is coarse grained, whereas the second one is very fine. In case of capacity, having a fine reading is acceptable because the capacity value is a snapshot of how much energy the battery is currently holding. On the other hand, a fine reading of *BatteryLifeTime* is unacceptable in our case because it fluctuates frequently based on the "this moment" discharge rate and capacity. For example, 10 consecutive readings of *BatteryLifeTime* with 1-second intervals can have an hour difference between some of the readings. As a result, we used *GetSystemPowerStatus* for *BatteryLifeTime* and *BATTERY_QUERY_INFORMATION* for battery capacity.

Collecting battery metrics at a very low time interval, such as every 1 ms, will give an accurate timeline of changes in power consumption. However, this method requires frequent polling of information, which incurs a high overhead. Since we adopted a coarse-grained reading of *BatteryLifeTime*, we profiled the battery life behavior in order to determine the accuracy of our approach and the ideal collection frequency. Profiling the battery life behavior consists of the following steps:

1. We disabled all power-management functionality of the platform power plan in order to maintain consistent power consumption of the platform.
2. We disabled all network adapters in order to avoid their periodical activities.
3. We fully charged the battery.
4. We disconnected the power cable.
5. We kept the platform in idle mode with the screen on.
6. We let the battery drain while collecting, at a time interval T, the remaining capacity and battery life remaining in seconds.

Figure 6.3 represents the data collected on a Surface 2 Pro tablet at a 3-second interval, and Figure 6.4 represents the data collected on a Dell Convertible tablet and Ultrabook at a 2-minute interval. By comparing the two figures, it becomes obvious that with a 3-seconds interval, there is higher fluctuation in the remaining battery life estimation graph compared to 2-minutes interval granularity. As result, we determined that a medium interval, such as a 3-second interval, resulted in an unclear picture because the reported values had high variance. On the other hand, collecting at a relatively high granularity (a 2-minute interval) let us collect the data with low variance and gave us consistent values, which let us be more accurate.

The other noteworthy observation is that when the remaining battery life reaches 5%, the platform enters hibernation mode. When we disable hibernation, as soon as the battery level reaches 5%, the capacity drops sharply, and within few minutes the platform switches off. As a result, the platform requires a significant amount of time on AC power before it can boot again. Therefore, we highly discourage disabling the hibernation setting when the battery level reaches 5%.

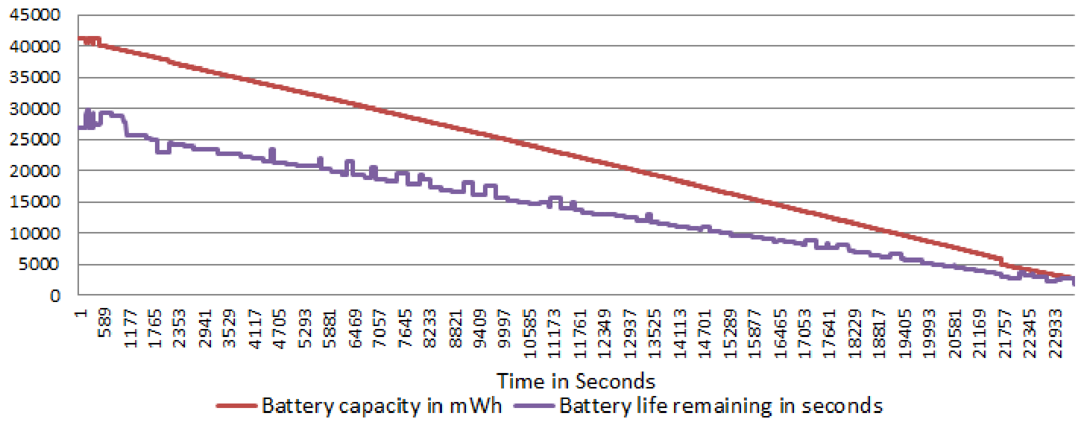


Figure 6.3: Relationship between battery capacity and remaining battery life over time at a 3-second interval on Surface 2 Pro tablet.

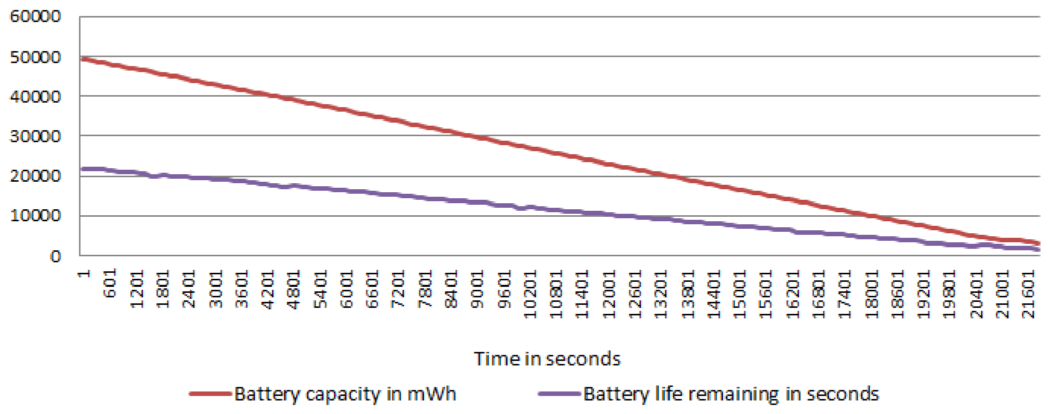


Figure 6.4: Relationship between battery capacity and remaining battery life over time at a 2-minute interval on Dell convertible.

6.5.3 Implementation

Our implementation is aimed at accomplishing the tool's objectives and is influenced by the preliminary studies performed using our target operating system. The following subsections describe in detail our technical implementation of each module.

Calibration Module

Self-modeling power consumption of a platform is highly dependent on the component collection it contains. Because these components vary from one platform to another, and even components from different vendors can vary their power consumption, it becomes important to build a self-modeling approach in order to estimate the power consumption of the platform in question. As a result, the calibration module aims to determine the power consumption of each device component. Once BatteryExtender is installed, users are required to run calibration at least once when the battery capacity is between 95 and 10 percent due to the fact that the discharge curve experiences a sharp drop when full and when it is almost empty, according to Tremblay *et al.* [96]. Moreover, users may choose to run the calibration periodically in order to improve accuracy of BatteryExtender due to the fact that the battery's resistance changes over time. The test duration is about 3 hours depending on the number of platform components. During this period, users are not allowed to use the platform.

Prior to calibrating, we need to change the platform power-management settings. Most (if not all) platforms support a power-management policy that suspends the hard disk when not in use and changes the processor frequency on the basis of the processor's load. We must disable both in order to keep the platform's power consumption constant during the calibration phase. In addition, we must be able to perform the following three commands:

- **Determining the platform's device components:** In order to determine the list of available devices, we use Plug and Play (PnP) configuration manager functions. We get a handler to device node *DevNode* and we iterate through them, looking for the list of devices. We can enable devices using the *CM_Enable_DevNode* function and disable them using *CM_Disable_DevNode*.

- **Determining the display brightness:** In order to get display brightness, we can create a handler to the LCD device, and then, using *DeviceIoControl*, we can change device percentage brightness according to the desired value.

- **Determining the display refresh rate:** In order to get a supported display refresh rate, we get a handler to the *DISPLAY_DEVICE* using *EnumDisplay-Devices*. Then, using *EnumDisplaySettings*, we can extract the display refresh rate supported using *dmDisplayFrequency*.

Finally, the calibration module automatically performs the following steps with the exception of the first step, which gives the user step-by-step instructions of how to perform it:

1. Disable the power management policy, which suspends HD and changes the processor's frequency.
2. Terminate all running applications with the exception of BatteryExtender.
3. Get the list of all devices and disable all of them.
4. Set the display brightness to 75 percent because we noticed when platforms are on battery, the power-management module changes the display brightness to 75.
5. Get the current display refresh rate, but keep it as its default setting.
6. Sleep for 120 seconds in order to avoid overhead from our changes.
7. Determine idle battery capacity consumption for 10 minutes by getting battery capacity at t_0, sleeping for 10 minutes, and then getting battery capacity at t_1. Then, we calculate the difference as shown in Equation 6.3.
8. Select one device from the list of PnP devices and enable it.
9. Sleep for 120 seconds.
10. Determine device battery capacity consumption for 10 minutes as described for the idle case.

11. Repeat the previous three steps until all devices have been profiled.
12. Change the display brightness to 25, 50, and 100 percent (one at a time) and then repeat steps 9 and 10.
13. If the display supports multiple refresh rates, change the refresh rate and then repeat steps 9 and 10.
14. Enable all devices.
15. Save the calibration values to an XML file.

Finally, by following this process, we can determine the power consumption of each device. Equation 6.4 shows the relationship of energy in joules to watts per second. Since our battery capacity is in milliwatts per hour, we can convert it to joules as shown in Equation 6.5. Then, we can determine the platform's average power consumption by applying Equation 6.6, where d is the duration in seconds. We conclude by calculating the power savings (gain) P$_{Saving}$ of the platform device by applying Equation 6.7, where P$_{Idle}$ is the power consumed during idle scenario, and P$_{Device}$ is the power consumed when the device was enabled or the display was set at a specific setting.

$$\Delta C_X = C_{Xt_0} - C_{Xt_1} \tag{6.3}$$

$$E_{(j)} = P_{(W)} * T_{(S)} \tag{6.4}$$

$$E_{(j)} = \Delta C_{(mWh)} \times \frac{1}{\underbrace{1000}_{\text{Convert to Watt}}} \times \underbrace{3600}_{\text{Convert to Seconds}} \tag{6.5}$$

$$P = \frac{E_{(j)}}{d_{(s)}} \tag{6.6}$$

$$P_{Saving} = P_{Device} - P_{Idle} \tag{6.7}$$

Energy Profiling Module

The energy profiling module consists of two parts. The first part is to energy profile the applications running on the platform, and the second is to energy profile the platform devices.

Energy Profiling of Applications: In order to determine the energy consumption of applications, we relied on the Machine Specific Registers (MSRs) of the System-on-Chip (SoC). Basically, the processor provides a variety of MSRs, which the processor uses to control and report processor performance. In order to be able to read them, the application must run at the kernel level (Ring0). In our implementation, we rely on the MSRs provided by Intel processors (processors with the code-name Ivy Bridge or later). We chose Intel processor's in particular because Intel currently dominates the market share for Windows platforms, with the exception of Windows RT platforms which use a different manufacturer of processors.

Intel processors support four nonarchitectural MSRs for Running Average Power Limit (RAPL) [15]. The first one is *MSR_RAPL_POWER_UNIT*. This register contains power units from bits (3:0), energy status units from bits (12:8), and time units from bits (19:16). The remaining ones are *MSR_PKG_ENERGY_STATUS*, *MSR_PPO_ENERGY_STATUS*, and *MSR_PP1_ENERGY_STATUS*, which report package, core, and graphics actual energy consumption. The MSRs are updated at approximately 1-ms intervals and the register wraparound time is about 60 seconds when power consumption is high.

In order to energy profile the applications, first we initialize the driver and read the power unit MSR determine the energy units. Then, at a 1-second interval, we collected the battery capacity, energy MSRs, and processes running on the platform. In order to get the list of processes, we got a snapshot of the processes and got a handler for each process. Through the handler, we were able to get the application name using *QueryFullProcessImageName*; then, using *GetProcessTimes*, we retrieved the processes' creation time, exit time, kernel time, and user time. In addition, using *NtQuerySystemInformation*, we collect *SystemProcessorPerformanceInformation*. Based on this information, determined the percentage of active time of the processor and each process. All metrics for processes belonging to the same application were combined together. In order to calculate the energy used by package, core, and graphics, we calculated the ΔE_{MSR} based on Equation 6.8 for each energy MSR and where U is the energy unit retrieved from *MSR_RAPL_POWER_UNIT*.

Then, in order to allocate per-application energy consumption, we first need to consider "package energy" versus "core energy". In our approach, we considered the package power consumption instead of just looking at the core power consumption. We based this approach on our previous work [71], which highlights the importance of aligning the utilization of cores in a multicore platform in order to allow the package to remain in a low idle-power state for the longest possible duration. More specifically, the platform in Figure 6.6 consumes less energy when compared to the platform in Figure 6.5, even though the percentage of CPU utilization and energy consumption of each core for the entire duration is the same in both scenarios. Since we are considering package energy, which includes, both the core energy and the graphics energy, we subtract the energy consumption of graphics from package. As a result, we can calculate the total energy used by the CPU as shown in Equation 6.9. Finally, we allocate the energy consumption of application X as E_X as shown in Equation 6.10, where U_X is the percentage of CPU usage of application X. We saved this information for each application in addition to the average Rate (power consumption) of the entire platform during that period of time which can be calculated using Equation 6.11.

We repeated this profiling technique for 50 iterations because CPU utilization of applications varies with time. One reading will not be enough to determine long-term effect on the overall battery life. However, with 50 iterations, we can have a better overview of the CPU utilization pattern of applications without posing extensive overhead of continuous polling of data. Upon completion, we got the average energy consumption of each application. We also calculated the average discharge rate. Then, in order to determine the battery life savings upon termination of the application Life$_{Saving}$, we used Equation 6.12, where we estimated the current battery life based on average discharge rate and we determine the savings by recalculating battery life based on average discharge rate minus the application's power consumption. Finally, we rank the top 5 most power-consuming applications.

Applications consume resources other than the processor, such as memory and disk. There are already many studies that can perform power profiling of memory and disk. For instance, MemPower [83] is a tool that can trace the memory usage and calculate power and energy consumption of the memory hardware. Dempsey [102] is a disk simulation environment that includes accurate modeling of disk power consumption. Despite the fact that we can collect per-process memory usage and disk usage, we do not add the per-process energy usage of either one (memory or disk) to the processor's power usage because our tool's goal is to be used without any external measurement tools. In addition, we do not want to limit its usability to a subset of available platforms. As a workaround, we provide BatteryExtender users with the extra option of adding as an input the known MemoryRead and MemoryWrite power consumption values and/or DiskRead and DiskWrite power consumption values. In that case, the tool will collect the memory and disk utilization values and factor them into the overall estimated energy consumption of each application. If this information is not provided, we disregard the values.

$$\Delta E_{MSR} = \{E_{t_0} - E_{t_1}\} * U \tag{6.8}$$

$$E_{All} = \Delta E_{Package} - \Delta E_{Graphics} \tag{6.9}$$

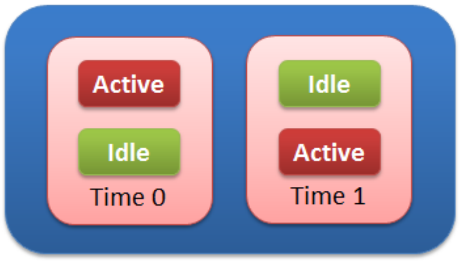


Figure 6.5: One core is active at each timestamp, resulting in an active package.

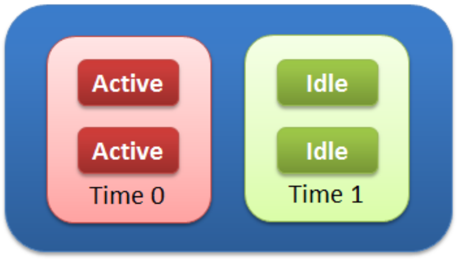


Figure 6.6: Two cores are active at timestamp 0, resulting in an active package, and are both idle at timestamp 1, resulting in an idle package.

$$E_X = E_{All} * U_X \tag{6.10}$$

$$Rate_{Avg} = \frac{\Delta C}{\Delta t} \tag{6.11}$$

$$\text{Life}_{Saving} = \frac{C}{Rate_{Avg} - P_{App}} - \frac{C}{Rate_{Avg}} \tag{6.12}$$

Energy Profiling of Platform Devices. In order to energy profile platform components, we use the XML file generated by the calibration module. Then, we iterate through all available devices in order to determine their state (active or disabled) in addition to checking the display brightness and refresh rate. Then, using Equation 6.13, we can calculate the life savings (or lost) Life$_{Saving}$, where P$_{Saving}$ is calculated based on Equation 6.7 and where $Rate_Avg$ is the same number as the one calculated during energy profiling of applications.

$$\text{Life}_{Saving} = \frac{C}{Rate_{Avg} - P_{Saving}} - \frac{C}{Rate_{Avg}} \tag{6.13}$$

User-Interactive Module and Power-Management Module

The user-interactive module is built using MFC Visual C++. Upon completion of the energy-profiling module, the user-interface in the user interactive module gets updated. The users will be able to see all the devices they can control and their estimated battery life savings. In addition, they will see the top 5 battery-consuming applications with their estimated battery life saving. Using a checkbox, they can select the devices they want to control and the applications they want to terminate. By confirming their selection, the power-management module is triggered to change the device's state as described in the calibration module based on the user's selection. In addition, using the process's ID, we can issue terminate process. Finally, the power-management module calculates the estimated battery life savings by adding up all the life savings values based on the changes confirmed by the user. Then, it calculates the minimum life savings duration between original duration selected by the user and the expected life savings calculated. Using this information, the tool can calculate the required battery duration Life$_{Req}$, as shown in Equation 6.14, where Life$_{Current}$ is the battery life prior to platform reconfiguration and Life$_{Min}$ is the calculated minimum life savings. Finally, it passes the value of Life$_{Req}$ to the monitoring module.

$$\text{Life}_{Req} = \text{Life}_{Current} + \text{Life}_{Min} \tag{6.14}$$

Monitoring Module

In order to start monitoring battery life, a new thread is created. At a 2-minute interval, it collects the expected battery life as described in section 6.5.2. It subtracts interval time i from Life$_{Req}$ as shown in Equation 6.15 and then compares the value to the expected battery life collected. If after five iterations, Life$_{Req}$ is less than expected battery life, the user is notified that the expected battery life is less than the required battery life. At this point, the user can either energy profile the platform again or terminate the monitoring module. If there are not five consecutive errors, the monitoring module continues to monitor until the Life_{Req} reaches 2 minutes.

$$\text{Life}_{Req} = \text{Life}_{Req} - i \tag{6.15}$$

Other Useful Features for Users

We also implemented three additional useful features:

- **Battery Usage Interface:** The battery usage interface enables the users to collect a log of battery metrics, including timestamp, capacity, discharge rate, voltage, and expected battery life. The collection interval is 30 seconds, and users can save the collected log to a csv file or delete the data and start over.
- **Interface for Power Profiling of Applications:** In our energy-profiling module, we collect the energy consumption of applications for only 50 seconds since we want to minimize the overhead. However, if a user wishes to power profile applications, they can use the interface for power profiling of applications. The interface enables users to see in real time the power consumption of applications using the method described in "Energy Profiling of Applications" section.
- **Battery Information Interface:** We also provide an interface that gives a detailed description of the battery such as its chemistry and wear level.

Finally, this summarizes our BatteryExtender implementation.

6.6 Experimental Analysis

Validating BatteryExtender consists of two parts: validating the tool in terms of reconfiguration of the hardware device components in order to save energy, and validating in terms of energy profiling of applications.

In order to validate our tool in terms of reconfiguration of the hardware device components, we ran a set of scenarios using the two platforms as described in Tables 6.1 and 6.2. The scenarios chosen are: download, video playback, and video streaming. Prior to running our experiments, we set the power-management policy to the default platform settings and terminated all (foreground and background) applications with the exception of BatteryExtender and the test application.

For each scenario, we ran two test cases. The first test case is the default case (DF), where we use the default platform settings. The second test case is the BatteryExtender case (BE), where users' commands are set to extend the battery life for "10 minutes." We chose "10 minutes" as opposed to a different duration because our goal was to determine the following: (1) whether can save battery by examining the amount of battery capacity saved, and (2) the accuracy of our tool based on expected capacity savings (based on the combination of choices selected) and actual capacity savings. In this test case, we changed the platform configuration based on the test scenario. For example, when Wi-Fi was not needed, we disabled it.

During all our experiments, we collected the battery metrics using our "Battery Usage" GUI as described in Section 6.5.3. We basically started collecting the battery metrics at the start of the test scenario, and upon completion, we stopped collecting the battery metrics and saved the log file.

During our experiments, we calculated **actual total capacity used** (energy used) by test case X denoted as $\Delta \mathrm{C_X}$ as shown in Equation 6.3 where C_{Xt_0} is the capacity at beginning of test case X and C_{Xt_1} is the capacity at the end of the same test case.

In addition, we calculated the **actual total capacity savings** (total saved energy) for scenario X denoted as AT_{sav_X} as shown in Equation 6.16 where we get the difference between total capacity used by DF test case and the one used by BE test case.

We also calculated the **expected capacity savings** for scenario X denoted as $\mathrm{E}_{\mathrm{sav_X}}$ as shown in Equation 6.17 where n is the total number of disabled or modified devices and $\mathrm{E}_{\mathrm{sav_D}}$ is expected capacity savings for device D for duration $\mathrm{d}_{\mathrm{base}}$ in minutes.

Moreover, we calculated the **total expected capacity savings** for scenario X denoted as $\mathrm{ET}_{\mathrm{sav_X}}$ as shown in Equation 6.18 where d_X is the total test duration in minutes.

$$AT_{sav_X} = \Delta C_{\mathrm{DF}} - \Delta C_{\mathrm{BE}} \tag{6.16}$$

$$E_{\mathrm{sav_X}} = \sum_{D=1}^{n} E_{\mathrm{sav_D}}, d_{\mathrm{base}} = 10 \tag{6.17}$$

$$ET_{\mathrm{sav_X}} = \frac{E_{\mathrm{sav}_X} \times d_X}{d_{\mathrm{base}}} \tag{6.18}$$

In order to validate BatteryExtender in terms of energy profiling of applications, we ran applications on the platform under testing. Then, using BatteryExtender, we profiled the energy used by the applications. Then, we terminated the applications using the User Interface Module and observed the impact on the platform's overall energy consumption.

The following subsections contain the calibration results in addition to detailed description and results for each scenario for validating the reconfiguration of device hardware components in addition to other case studies related to energy consumption of mobile devices. We also present the results for validating BatteryExtender in terms of energy profiling of applications.

6.6.1 Calibration Results

First of all, we ran the calibration module on each device. Table 6.4 represents the values collected for Dell Convertible, and Table 6.5 represents the values collected for Surface 2 Pro. All devices in the table are self-explanatory with the exception of ”USB Root Device (xHCI)” in Table 6.4. By disabling this device, we disable USB input, HID Sensor Collection, the camera, and the Bluetooth adapter. Based on these results, it is evident that the same components, on different platforms, can consume different amounts of battery capacity.

Device Dell Convertible	**Capacity used in 10 minutes**	**Capacity saved**
Idle @ 75 Brightness	1070	0
WiFi	1190	120
Bluetooth	1099	29
NFC	1075	5
HID Sensor Collection	1080	10
VS Location Simulator	1080	10
Touchscreen Sensor	1080	10
Audio	1080	10
Camera	1090	20
Printer Queue	1080	10
USB Root Device (xHCI)	1150	80
Refresh @ 59 Hz	1050	20
Brightness @ 25	880	190
Brightness @ 50	990	80
Brightness @ 100	1180	-(110)

Table 6.4: Calibration results for Dell Convertible.

6.6.2 Download Scenario

The first scenario is the download scenario. During this scenario, we used Amazon Unbox Video Player [9] to download a previously purchased movie. We chose to download on each platform ”Despicable Me” in HD, where the movie size is 1.91 GB.

Device: Surface 2 Pro	**Capacity used in 10 minutes**	**Capacity saved**
Idle @ 75 Brightness	740	0
WiFi	792	52
Bluetooth	769	29
HID Sensor Collection	748	8
VS Location Simulator	748	8
Pen Sensor	747	7
Touchscreen Sensor	747	7
Audio	752	12
Camera Rear	745	5
Camera Front	745	5
Printer Queue	750	10
Brightness @ 25	604	136
Brightness @ 50	670	70
Brightness @ 100	947	-(207)

Table 6.5: Calibration results for Surface 2 Pro.

For the Default (DF) test case, we started BatteryExtender. Then, using the ”battery usage” UI, we started collecting the battery capacity remaining at a 30-second interval. Next, we started Amazon Unbox Video Player, selected the movie, and pressed on download. Upon completion of download, we stopped collecting ”battery usage” and saved the log.

For the BatteryExtender (BE) test case, we started BE and set 10 minutes for extension duration. Then, we selected the following metrics as shown in Table 6.6 for Dell Convertible and in Table 6.7. Next, using the ”battery usage” UI, we started collecting the battery capacity remaining at a 30-second interval. We then started Amazon Unbox Video Player, selected the movie, and pressed on download. Upon completion of download, we stopped collecting ”battery usage” and saved the log.

The results comparing DF versus BE test cases for Dell Convertible are displayed in Figure 6.7. A major issue was observed in this scenario. The download duration during default settings took 3750 seconds (1 hour, 2 minutes, and 30 seconds), whereas the download duration during BE scenario took 1080 seconds (18 minutes), 71.2 %

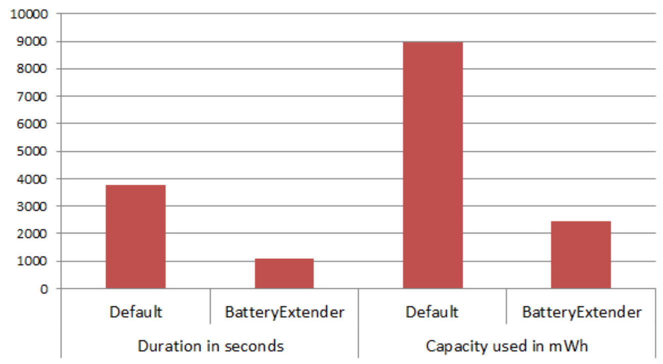


Figure 6.7: Battery capacity over time during download scenario for Dell Convertible.

faster than default test case. In addition, the ΔC_{DF} is 8980, whereas the ΔC_{BE} is 2440. The AT_{sav} is 6540, for a total of 72.83% savings. These results far exceeded our expectation and at first glance appeared to be abnormal.

In order to determine the cause of this huge savings, we conducted further analysis. We determined that by enabling "USB Root Device (xHCI)," the download speed as shown by the application drops from an average of 17.4 Mbps to 4528 Kbps. In addition, the CPU utilization jumps from an average of 10% to 18%. Moreover, the average memory utilization jumps from an average of 32% to 55%, and the average cache of 558 MB to gradually reaching 1.6 GB. The cause of this change is due to the "Network Security Service," which we managed to disable when disabling "USB Root Device (xHCI)." Network security is definitely an important feature, but increasing the download time by 71.2% is unacceptable to most mobile device users. So, it is definitely an issue to be examined.

The results comparing DF versus BE test cases for Surface 2 Pro are displayed in Figure 6.8. The ΔC_{DF} is 2211, whereas the ΔC_{BE} is 1865. The test duration was 16.5 minutes for both test cases. The AT_{sav} is 346, for a total of 15.65% savings. Our ET_{sav} is 374.55, which results in a 92.37% accuracy rate.

Disabled devices	**Expected capacity saving in 10 minutes**
NFC	5
VS Location Simulator	10
Touchscreen Sensor	10
Audio	10
Printer Queue	10
USB Root Device (xHCI)	80
Refresh @ 59 Hz	20
Brightness @ 25	190
Expected Savings in 10 minutes	**335**

Table 6.6: Disabled devices and display settings associated with expected capacity savings during download scenario for Dell Convertible.

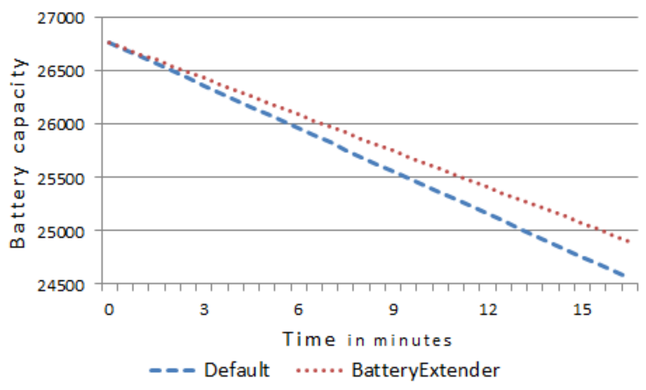


Figure 6.8: Battery capacity over time during download scenario for Surface 2 Pro.

6.6.3 Video Playback Scenario

The video playback scenario consists of watching a movie using Amazon Unbox Video Player. We played ”Despicable Me” in HD. The movie duration is 95 minutes.

We ran both test cases DF and BE and collected the battery metrics as described in the previous scenario. We selected the following metrics to disable (change in the case of display), as shown in Table 6.10 for Dell Convertible and in Table 6.9 for Surface 2 Pro.

The results comparing DF versus BE test cases for Dell Convertible are displayed in Figure 6.9. Energy consumed during DF ΔC_{DF} is 15,450, whereas the ΔC_{BE} is

Disabled devices	**Expected capacity savings 10 minutes**
Bluetooth	29
HID Sensor Collection	8
VS Location Simulator	8
Pen Sensor	7
Touchscreen Sensor	7
Audio	12
Camera Rear	5
Camera Front	5
Printer Queue	10
Brightness @ 25	136
Expected Savings in 10 minutes	**227**

Table 6.7: Disabled devices and display settings associated with expected capacity savings during download scenario for Surface 2 Pro.

12,327. The AT_{sav} is 3,123, for a total of 20.21% savings, whereas the ET_{sav} is 3,087.5. Based on these results, the accuracy rate is 98.86 %. In this case, despite the fact that we disable "USB Root Device (xHCI)," our accuracy rate remained high because Wi-Fi was disabled.

The results comparing DF versus BE test cases for Surface 2 Pro are displayed in Figure 6.10. The ΔC_{DF} is 9,435, whereas the ΔC_{BE} is 8,194. The AT_{sav} is 1,241, for a total of 13.15% savings. Our ET_{sav} is 1,244.5, which results in a 99.72% accuracy rate.

6.6.4 Video Streaming Scenario

The video streaming scenario consists of watching a movie using YouTube. The movie selected is "Elephant Dreams in HD" playing for 10 minutes. We ran both test cases DF and BE and collected the battery metrics as described in the previous scenario. We selected the following metrics to disable (change in the case of display), as shown in Table 6.10 for Dell Convertible and in Table 6.11 for Surface 2 Pro.

Disabled devices	**Expected capacity savings in 10 minutes**
WiFi	120
NFC	5
Touchscreen Sensor	10
Printer Queue	10
USB Root Device (xHCI)	80
Refresh @ 59 Hz	20
Brightness @ 50	80
Expected Savings in 10 minutes	**325**

Table 6.8: Disabled devices and display settings associated with expected capacity savings during video playback scenario for Dell Convertible.

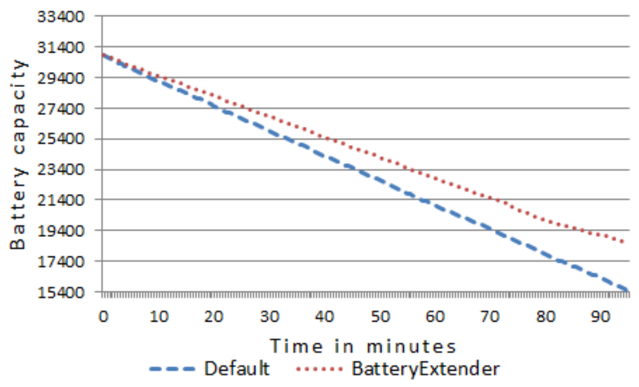


Figure 6.9: Battery capacity over time during video playback scenario for Dell Convertible.

The results comparing DF versus BE test cases for Dell Convertible are displayed in Figure 6.11. Energy consumed during DF ΔC_{DF} is 3080, whereas the ΔC_{BE} is 2,170. The AT_{sav} is 910, for a total of 29.54% in energy savings, whereas the $ET_(sav)$ is 225.5. In this case, even though "USB Root Device (xHCI)" was disabled, we were able to watch the movie without any time waiting for buffering. However, the CPU and memory activities again spiked due to the security service. As a result, the percentage of energy savings far exceeded our expectations but wasn't as significant as in the download scenario.

The results comparing DF versus BE test cases for Surface 2 Pro are displayed in

Disabled devices	**Expected capacity savings 10 minutes**
WiFi	52
Bluetooth	29
HID Sensor Collection	8
VS Location Simulator	8
Pen Sensor	7
Touchscreen Sensor	7
Camera Rear	5
Camera Front	5
Printer Queue	10
Expected Savings in 10 minutes	**131**

Table 6.9: Disabled devices and display settings associated with expected capacity savings during during video playback scenario for Surface 2 Pro.

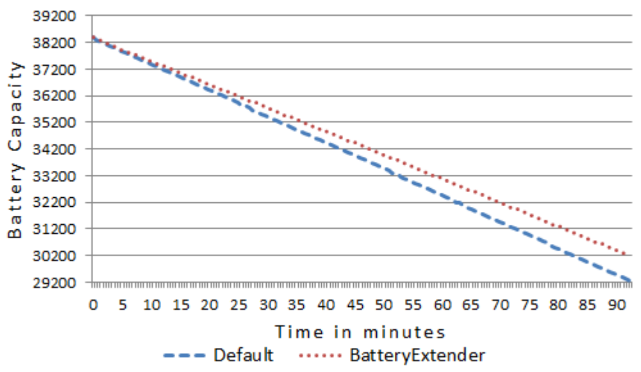


Figure 6.10: Battery capacity over time during video playback scenario for Surface 2 Pro.

Figure 6.12. The ΔC_{DF} is 1,694, whereas the ΔC_{BE} is 1,524. The AT_{sav} is 170, for a total of 10.03% savings. Our ET_{sav} is 163.9, which results in a 96.41% accuracy rate.

6.6.5 Other Case Studies

We also performed other case studies to evaluate some functionality utilized by users in order to determine their impact on battery drainage. More specifically, we evaluated the impact of using touchscreen versus the keyboard and the impact of changing device orientation on energy consumption.

Disabled devices	**Expected capacity saving in 10 minutes**
NFC	5
Touchscreen Sensor	10
Printer Queue	10
USB Root Device (xHCI)	80
Refresh @ 59 Hz	20
Brightness @ 50	80
Expected Savings in 10 minutes	**205**

Table 6.10: Disabled devices and display settings associated with expected capacity savings during video streaming scenario for Dell Convertible.

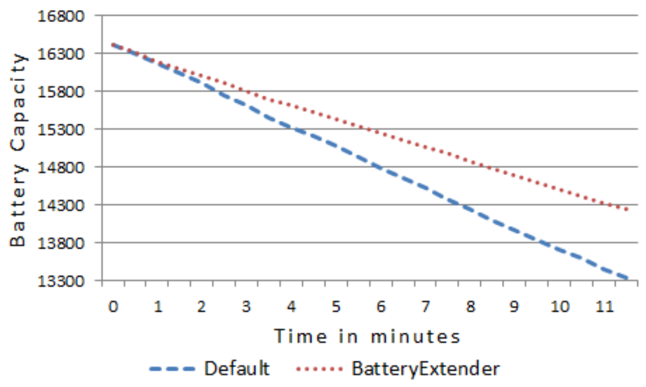


Figure 6.11: Battery capacity over time during video streaming scenario for Dell Convertible.

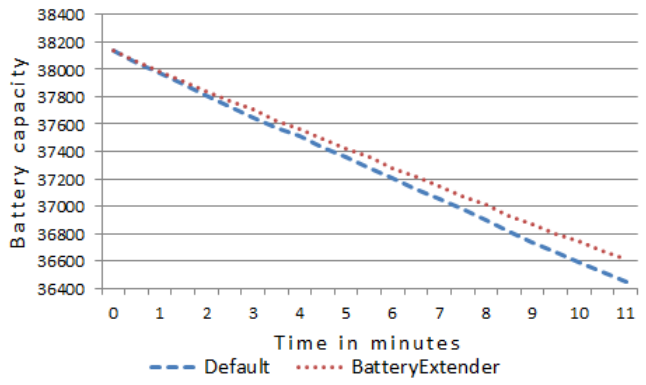


Figure 6.12: Battery capacity over time during video streaming scenario for Surface 2 Pro.

Disabled devices	**Expected capacity savings 10 minutes**
Bluetooth	29
HID Sensor Collection	8
VS Location Simulator	8
Pen Sensor	7
Touchscreen Sensor	7
Camera Rear	5
Camera Front	5
Printer Queue	10
Brightness @ 50%	70
Expected Savings in 10 minutes	**149**

Table 6.11: Disabled devices and display settings associated with expected capacity savings during during video playback scenario for Surface 2 Pro.

Touchscreen vs. Keyboard

This case study was performed in order to evaluate the impact of using touchscreen versus keyboard on the energy consumption of mobile devices. We used Dell Convertible, which comes equipped with a built-in keyboard and has a 10-touchpoint touchscreen. During this case study, we browsed 100 pictures by double clicking (tapping in the case of touch) on a picture, viewing it in the Windows 8 default picture viewer, swiping from the right edge of the screen, clicking (tapping) on the Start option, clicking (tapping) on Desktop, and then double clicking (tapping) on the next picture. We repeated these steps until we viewed all 100 pictures. The test duration was 10 minutes, during which we collected the changes in capacity similar to the previous test scenarios. Figure 6.13 represents the battery capacity over time during the touchscreen versus keyboard test scenario. Based on our results, during the touchscreen test case, ΔC_{Touch} is 1480 mWh whereas $\Delta C_{\text{Keyboard}}$ is 1590 mWh during keyboard test case. As a result, by using touchscreen for commands instead of the keyboard, we can save 6.91% in energy.

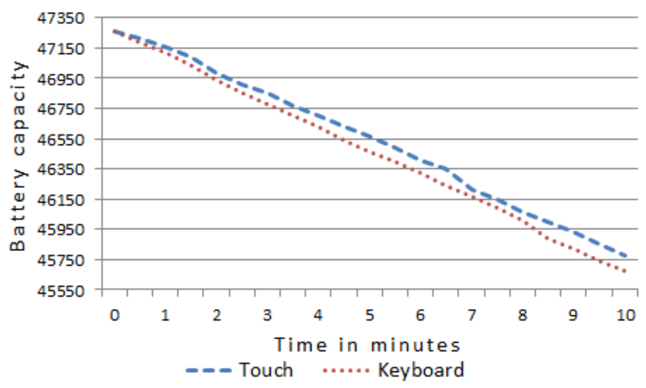


Figure 6.13: Battery capacity over time during touchscreen vs. keyboard test cases for Dell Convertible.

Impact of Changing Device Orientation on the Energy Consumption

This case study was performed in order to evaluate the impact of changing device orientation on the energy consumption of mobile devices. We used Surface 2 Pro during this case study. We ran an "Elephant Dreams HD" movie using YouTube. We ran the test for 5 minutes, during which we rotated the device 90 degrees 15 times per minute. Figure 6.14 represents the battery capacity changes over time for the test case where we changed the landscape and the test scenario without changing landscape. We consumed 763 mWh when we kept the device in the same orientation and 871 mWh when we changed the orientation. As a result, changing the device orientation cost the user 12.39% more energy.

6.6.6 Validating Energy Profiling of Applications

In order to validate BatteryExtender in terms of energy profiling of applications, we used Surface 2 Pro. We chose to extend battery life for "10 minutes" for the same reason as described when validating BatteryExtender in terms of platform reconfiguration. BatteryExtender was able to detect "Symantec Antivirus" running in the background with 22.14% CPU utilization and an estimated 2,331 mW of power usage.

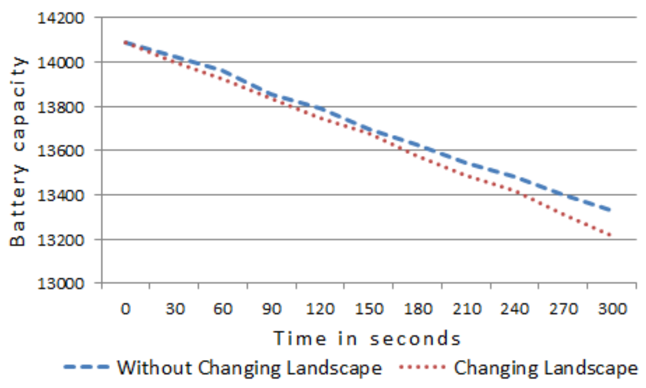


Figure 6.14: Battery capacity over time during effect of landscape change for Surface 2 Pro.

Using battery usage interface, we collected the battery discharge rate (power) prior to terminating the app and the collected it again after 30 seconds after terminating the application. We noticed that the discharge rate dropped by 2,562 mW after terminating the application. As a result, the accuracy rate was 90.98%. Similarly, we repeated the validation steps, but we selected a different application to terminate. The application selected was Google Chrome, which was video streaming a YouTube video. Based on BatteryExtender, YouTube was utilizing 2.28% of CPU usage consuming 555 mW. Upon termination, we noticed that the discharge rate dropped by 608 mW. As a result, the accuracy rate was 90.46%. Finally, it is clear that, using the current implementation of BatteryExtender we can power profile applications with relatively high accuracy rate despite. Since we are not considering memory or disk power consumption, our estimation of battery savings can be conservative. However, this technique still satisfies BatteryExtender's goals.

6.7 Conclusion and Future Work

In this paper, we presented BatteryExtender, a tool that enables users to extend battery life on demand. It enables users to reconfigure the mobile devices in order to utilize only the resources required for their specific tasks. It also provides an estimate of the impact of applications on the overall battery life. Using BatteryExtender, we were able to reduce energy consumption between 10.03 and 20.21 percent, depending on the workload. The accuracy rate ranged between 92.37 and 99.72 percent. In addition, in some rare cases, we were able to reduce energy consumption by 72.83 percent due to the platform's inefficient security service. In the future, we are planning on improving our resource power-consumption estimation by continuously profiling the platform when battery life is not an important resource for the user.

CHAPTER 7: ENERGY EFFICIENCY OF SYSTEM-ON-CHIP DEVICES

7.1 Introduction

It’s no secret that smartphones and tablets are increasingly becoming the ubiquitous choice of computing up to the point where they are affecting the sales of PCs. The number of Internet-connected devices is expected to reach 25 billion by 2015 and 50 billion by 2020 [27]. Users of these mobile devices believe in the “always connected” and “anywhere computing” paradigm. They want to be connected to their emails and their social media outlets (Facebook, Twitter) anytime and all the time. They want to be able to watch their movies or make Skype calls with the highest-possible definition. Of course, they also want to be able to download/upload their YouTube videos and pictures as fast as possible. Despite all of these wants, users still mandate a long battery life. They even expect a battery life comparable to that of their traditional-feature phones.

One major obstacle to the extension of these devices’ battery life is their strict weight and handheld property, which prevents the extension of the battery size. As a result, the extension of battery life becomes limited on the ability to optimize the battery usage of applications using the underlying hardware and the optimization of the battery usage of the hardware.

All the above user requirements and the devices’ physical requirements dictate how mobile System-on-Chip (SoC) vendors design their overall system: suddenly, scalability and low power are factors that veto any feature upgrade or changes to the design. Thus, the combination of hardware, software, and connectivity makes a platform's energy efficiency extremely important.

As difficult as it looks, SoC vendors are now increasingly employing the concept of "offloading" to their designs to address the twin problems mentioned above. Designs today integrate special-purpose digital signal processors (DSPs) and accelerators into the design and enable software developers to use these tiny engines.

To increase these SoCs' energy efficiency, we must understand their power consumption. Because SoCs comprise a large number of subcomponents, each contributing to the overall power consumption, it becomes necessary to consider those components when power profiling the devices, especially considering that the IP components are often responsible for a substantial portion of the SoC's power consumption. Therefore, it is important to power model and consider their effect in power-aware SoC design.

There are already many software- and hardware-based powerprofiling techniques that range between fine- and coarse-grained granularity. For the most part, system-level power analyses have focused on the processor, memory hierarchy, and interconnect when power profiling a system. Relatively little work has been done for modeling the power consumption of IP components and their impact on system-level trade-offs. Thus, despite the usefulness of the current profiling tools, there is still room for improvement because the nature of SoC devices requires even-finer sophisticated measurement tools in order to further optimize Intellectual Property (IP) block offloading.

7.1.1 Contribution

We make the following contributions in this paper:

- We highlight the importance of offloading and show how it can be effective in energy optimization for SoC devices. We provide supporting data for our claims by including thermal images of an SOC while offloading was enabled and while it was disabled.

- We make a strong case for new power profiling tools that take a holistic view of the systems, including peripherals and accelerators that are beyond the CPU. We provide two case studies, one using GPU/CPU for video decoding and one using DSP/CPU for audio decoding, to show that today's SoC devices require very fine and sophisticated power profiling tools to account for the SoC's exercised offloading mechanism of functionality to different IP blocks.

- We show that current software-based power profiling techniques for SoCs can provide an error rate close to 12%. Thus, they cannot be used for increasing the energy efficiency of workloads, which offload from CPU to the dedicated IP blocks.

7.1.2 Organization

This chapter is organized as follows: In Section 7.2, we give an overview of SoCs, including challenges and advantages. In Section 7.3, we explain the art of offloading and show its advantages. In Sections 7.4 and 7.5, we present both a profiling methodology for SoCs and two case studies: one power-profiling the SoC when utilizing the Graphics IP unit and one utilizing the Audio IP unit. Finally, we provide analysis and our conclusions in Section 7.6.

7.2 Overview of System-on-Chip

The SoC paradigm is currently the dominating architecture within mobile devices. The term "system-on-chip" (SoC) represents two things: the physical architecture of the product and the methodology used to design it.

- **SOC Physical Architecture:** SOCs integrate several subsystems, where traditionally many or all of which would have been separate discrete chips, into a single chip. The SoC may be a single silicon die, or possibly many dies inside a single package. Either way, an SoC is rarely the entire system on that single chip, but it usually encompasses the device's computing functions. Typical SoC subsystems include: the CPU (with one or many cores), memory, input/output (I/O), and storage, in addition to media such as video and audio, graphics, and camera.

- **SOC Methodology:** The SoC building methodology comprises two independent phases: the independent building of modular IP blocks, which represents the SoC's subsystems, and the integration of the IP blocks into a specific product, transforming all the components into a single integrated product.

  This SoC building methodology is necessary because each device utilizing an SoC defines the SOC requirements on the basis of the intended use of the device. Therefore, we cannot use the same SoC for a smartphone and a tablet. Both devices may share the same CPU cores, but the smartphone's SoCs might utilize a Digital Single Length Reflex (DSLR) camera functionality while the tablet may include more sophisticated graphics and media for watching high-definition movies and playing 3D games.

Figure 7.1 displays the overall architecture for Intel's Medfield SoC platform with an Intel Atom processor. The SoC contains many hardware accelerators such as a dedicated Image Signal Processor (ISP) for high-performance imaging, a PowerVR SGX540 2D/3D hardware engine for high-performance graphics and games, and a special low-power Audio Digital Signal Processing (DSP) for voice applications. Based on Figure 7.1, it is very clear that the bulk of the SoC is made up of special-purpose accelerators.

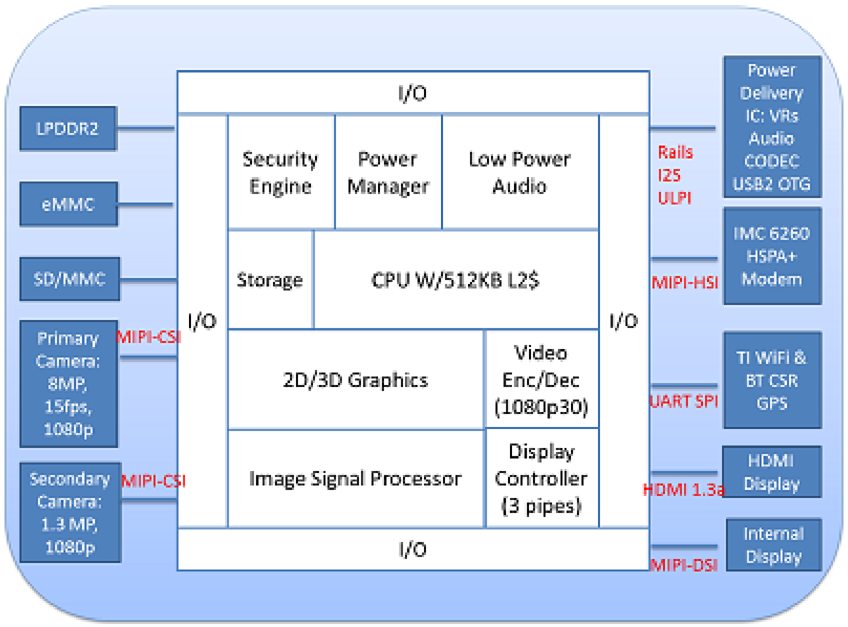


Figure 7.1: Medfield SoC Block Diagram - Penwell SOC (Intel Hi-K 32 nm Process Technology

7.2.1 Challenges of SoCs

Developing SoC devices is challenging for a variety of reasons, including integration of third-party IP modules, low-cost design and manufacturing requirements, limited hardware resources, and highly constrained power budgets.

Power issues are traditionally considered hardware problems, while software focuses on features and flexibility. However, power consumption is highly software-dependent and does not necessarily correlate well to thermal design power (TDP) specifications [42]. Therefore, hardware and software developers should be aware of energy consumption and seek to co-design SoC devices, to further improve the power consumption.

Traditional hardware and software co-design techniques focus on static partitioning or allocation of system resources to hardware and software, to implement specific applications such as video or audio encoder. As more functionality gets integrated into SoCs, traditional partitioning is not sufficient to optimally run multiple applications that have different performance, power, and thermal requirements. This motivates looking at new approaches to target overall power/thermal requirements, for energy-efficient SoC devices.

So far, the energy-awareness response has been focused on dynamic allocation within the CPU. The energy-aware processor delivers high performance when needed while consuming minimum active and idle energy when the CPU is not active [85]. Because IP blocks reside in a lower layer of the system than hardware components, traditional coarse-grained power management cannot accurately control the power states of IP blocks. For example, when a user touches the screen, several components will remain active to handle this user interaction. However, from the angle of IP block, some of the IP blocks are not used for handling tasks, and thus could be turned off to save energy. Secondly, because the current power-management system does not use software information, and manages hardware power states in a passive way, a delay between satisfying the conditions and updating hardware power states is inevitable. Because applications' activities have a direct relationship with hardware power status, software information could be used to aggressively set up the power states of hardware and improve the system's energy efficiency.

7.3 The Art of Offloading

It is very common (and right) to offload a task from the main processor to a specialized custom engine. The Application-Specific Standard Processor (ASSP) fraternity goes to the extreme of stitching up a totally fixed function engine to realize the best throughput efficiency and usage of the engine for a given workload. It is indeed true that a fixed-function compute engine unit for a Discrete Cosine Transform (DCT) is always more energy efficient and better throughput than running the DCT on a CPU.

One important point that often gets missed is the importance of the rest of the SoC components when tasks are offloaded. So far, because the CPU is the critical owner for any ”compute” process, the CPU decides the state of resources such as the SoC interconnect, memory controllers, and the path to memory. Whenever the CPU is being used, most of these high-power connections are turned on, which can equally add to the entire platform power consumption.

7.3.1 Advantages of Offloading

Integrating custom, special-function accelerators is one of the most complex exercises in low-power SoC architecture. Yet, designers willingly take up this challenge in return for the power benefits that these units provide, including more-efficient computing and an optimized path to memory.

Efficient Computing. Many multimedia operations such as video and audio processing, for example, rely on signal-processing concepts. Fundamental DSP algorithms, such as filters and transforms, depend on very fast multiply-and-accumulate operations. Furthermore, very low-latency outputs in a video frame or audio sync interval require the most efficient parallel processing between multiple execution units. Several architectures, including very long instruction word (VLIW) and single instruction, multiple data (SIMD) operations, support such capabilities. Fixed-compute function power budgets are much lower using a fixed-functionality engine than using a general-purpose CPU.

In order to highlight the importance of offloading, we used a thermal imaging tool in two scenarios: first, we ran audio playback without offloading, as Figure 7.2 shows, then we ran audio playback with audio DSP offloading, as Figure 7.3 shows.

Figures 7.2a and 7.3a represent the Medfield SoC with each component’s location; Figures 7.2b and 7.3b show the thermal image of this SoC.

Figure 7.2 shows audio playback without audio offloading. The temperature increases from blue to green and from green to red as shown in the thermal scale in Figure 7.4. Based on the thermal scale, the CPU is active, because it has the color green and all the remaining non-blue areas show the transistors are active as well.

Figure 7.3 shows audio playback with offloading and its corresponding thermal image. The thermal image shows the CPU switched off and the execution is running on the audio DSP. Many parts of the SoC are either shut off or optimized for low power consumption; indeed, almost the entire SoC is switched off, barring a few green spots indicating where executions are still running.

In Figure 7.2b, most IP blocks on the SoC are green because we aren't managing power efficiently. In contrast, in Figure 7.3b, most of the parts are either in low power or completely shut off, such as the Image Signal Process Unit, because we are not processing image signals and the CPU isn't in use. However, the video encoder/decoder IP block is green because we are offloading to it. The power management unit is also green because it needs to manage the power on the SoC. With these optimizations, the platform's capability for low-power audio playback increases the battery life to approximately 120 hours.

Optimized Path to Memory. The SoC can also save power by optimizing the path to memory from offload engines. A fixed-function accelerator has a finite bandwidth capability; therefore, SoC designers can reduce the interconnect speeds to save power. In addition, because operations are performed on the accelerators, it is possible to turn off the CPU and put the SoC in a special low-power state.

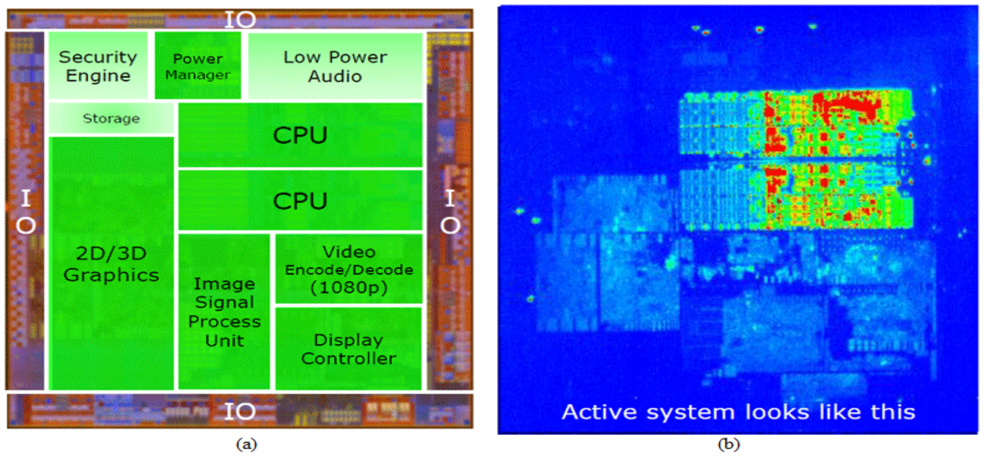


Figure 7.2: Medfield thermal image during audio playback without offloading.

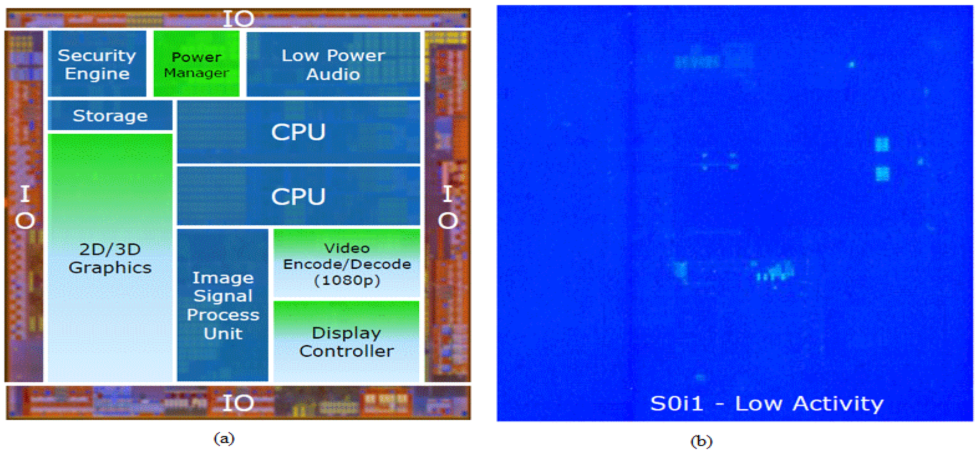


Figure 7.3: Medfield thermal image during audio playback.

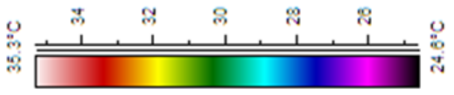


Figure 7.4: Thermal scale.

7.4 Profiling Methodology

We used different hardware to evaluate the fine-grained energy efficiency of an IP unit on the SoC. One of the test systems is called "Host," where data were collected. Host is connected to National Instrumented Data Acquisition (DAQ), which is calibrated for collecting accurate voltage as low as 1 mwatt and 1000 samples/sec [57]. The Active Signal Conditioning Card (SCA) is an analog signal conditioning front end for the National Instruments DAQ. The board has been designed to filter out noise typically found in a client system environment. DAQ has been configured to monitor several rails (each unit on the device under test) simultaneously and has an analog-to-digital converter with channel multiplexer. Another part of DAQ is connected to the "Target," which is the system under test. The system under test is based on the most current Intel 4th Generation processor low-processor SKU (Haswell). This processor features dual cores with multithreading and integrated graphics running a Windows 8 operating system [58]. The system is running on DC mode and a frequency of 0.8 MHz when idle. Figure 7.5 shows a detailed setup of the system.

"Host" collects and calculates the power by post-processing the channel input from DAQ. Each channel collects the data at a specified interval. In order to ensure a

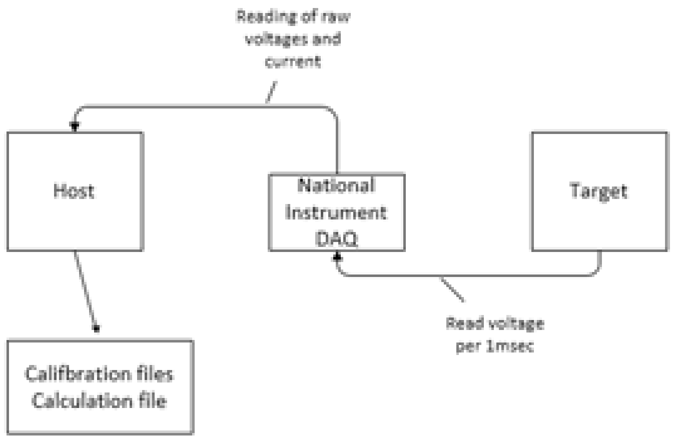

Reading of raw voltages and current
Host
National Instrument DAQ
Target
Read voltage per 1msec
Califbration files Calculation file

Figure 7.5: Power instrument flow diagram.

fair evaluation of different IP units, we calculated the delta of savings instead of the obsolete power number and ran each workload for 3 times as long to remove the variances of idle power. Figure 7.6 shows the delta variance between each run. The x-axis is tested by turning background activities off or on. As we moved from Test 1 to Test 3, there was a significant decrease in variance. Because we were looking for very fine granularity of power to find impact on IP unit, we needed to make sure variance between run-to-run is less than 0.05%. With Test 3, we achieved very low variance for package power which is SoC power, memory power, and audio device power on the platform.

Moreover, the instrumented system is fully functional, safe, and validated using high-end volt meter. The reason we chose to use a DAQ instead of other available software estimation tools is referred in Gurumurthi *et al.* [50], Brooks *et al.* [31], and Vijaykrishnan *et al.* [98]. The use of DAQ is crucial in this experiment since we not only focus on why IP units are important but we also show mW savings due to IP units and the possible techniques for IP units optimization. All of these goals definitely require significantly less variance and high granulating readings than any other available software-based profiling tool.

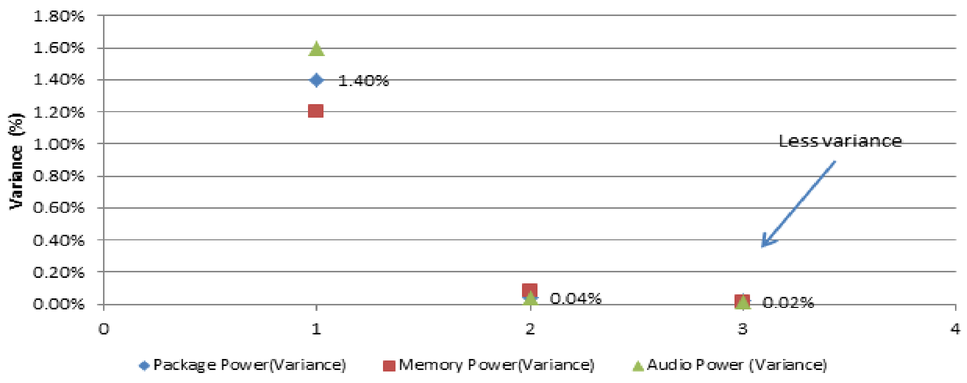


Figure 7.6: Variance of different OS configurations.

7.5 Experimental Results

In this section, we present two case studies that use IP offloading. The first case study evaluates the Graphics IP Unit, and the second case study evaluates the Audio IP Unit. Both case studies used the profiling methodologies as described in Section 7.4.

7.5.1 Case Study 1: Evaluation of Graphics IP Unit

Many studies have been published on the use of graphics offload versus running on a CPU. Streaming media power saving due to graphics unit offload has been discussed in the Intel Media Software Development Kit [91]. In this chapter, we build on the studies done in the streaming media field and extend them to optimize how the IP unit can be used and measure the power savings. In addition, this study shows usage of DAQ where we can measure small power gains versus software-based power estimation.

We run 1080p high-definition clip on a popular media player. We used the Blender foundation-based Elephant dream 1080p H264 clip for our experiment [41]. We tuned the software to either use hardware offload using graphics or only use software via the source code. The pseudocode in Figure 7.7 shows the call to be made to either do software or hardware processing.

```
sessionInit(SW/HW/AUTO)//soft/hard/auto detect
        while (bitstream) {
                decodeDecodeFrameAsync(bitstream,       [out]
                frame_d, [out] sync_d);
                coreSyncOperation(sync_e);
        }
        decodeClose(); // Close down
        sessionClose(); // Close session
```

Figure 7.7: Pseudocode for software and hardware offloading.

First, we ran the tests without utilizing IP offloading. In order to validate if the application is not using hardware IP offload, we used Microsoft GPUView tool to find the wakeup activities in graphics [53]. Figure 7.8 shows that the GPU hardware queue is empty while the application was doing media on local playback. Figure 7.9 shows the CPU utilization had been blocked during the processing, which means that the application was not performing any IP offload.

Then, we ran the test with the utilization of IP offloading. We also used software-based optimization and offloaded the content from CPU to graphics IP block. Figure 7.10, displays the impact of GPU offload on the CPU. The decode queue has been offloaded to the GPU IP, and the state of the CPU is changed to low power. Note that the processing of the frame remains the same, but owing to the use of IP block, the processing becomes more efficient and high performance.

Running the DAQ instrumentation on the platform before and after using the IP block let us notice significant power savings. Figure 7.11 represents the delta of power savings when using offloading compared to without offloading.

Despite these power savings while using offloading, there is still room for optimization. There is a need of optimizing the GPU IP pipeline and GPU calls from software. We used Microsoft GPUView to analyze the calls made to GPU and optimize to gain mW of savings [53]. Figure 7.12 shows the CPU/GPU unalignment during the offload. CPU activities are when an application is running on the CPU and ready to submit work to the GPU, while GPU activities shows the call after the CPU has submitted the job to the GPU. During the same duration, the CPU is blocked and waiting for the GPU to complete the job. The solution is to overlap GPU/CPU activities and unblock the CPU while the GPU is doing render processing.

Even though offloading to IP units gives power savings in watts, there's still significant room for improvement which can go undetected using software-based estimation tools. Internal studies have shown that current software-based estimations have huge variance with a small power delta. Using DAQ, we found the impact of power in a few hundred milliwatts due to unalignment and CPU/GPU concurrency issues.

Figure 7.14 shows the results in delta of power saving measured from an internal software tool with approximately 90% accuracy compared to DAQ. Table 7.1 shows that as the optimization opportunity becomes smaller, the error of measurement increases, and thus there is need for DAQ to get the low-power saving of specific IP units.

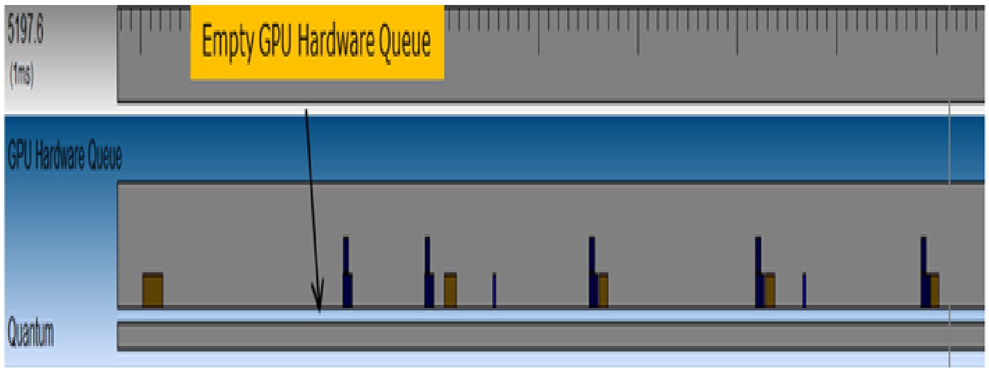


Figure 7.8: GPUView with empty queue.

7.5.2 Case Study 2: Evaluation of Audio IP Unit

Analyzing the SoC power is very important. There are already software estimation tools to provide the delta for huge power savings, but in today's era of computing

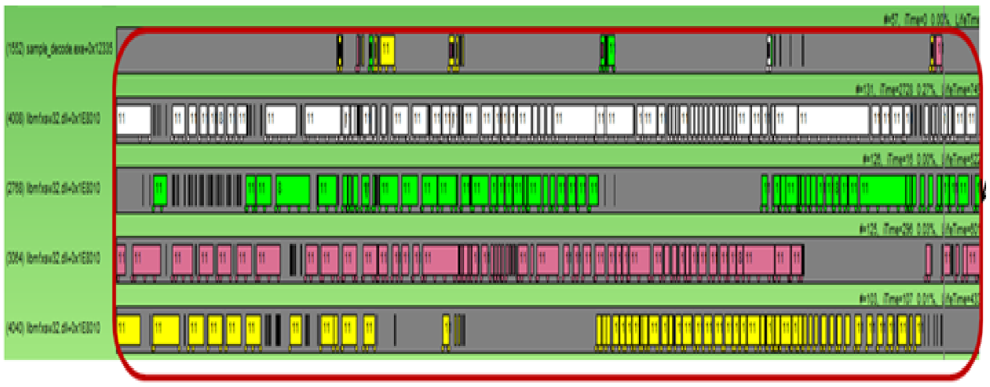


Figure 7.9: CPU activities during playback.

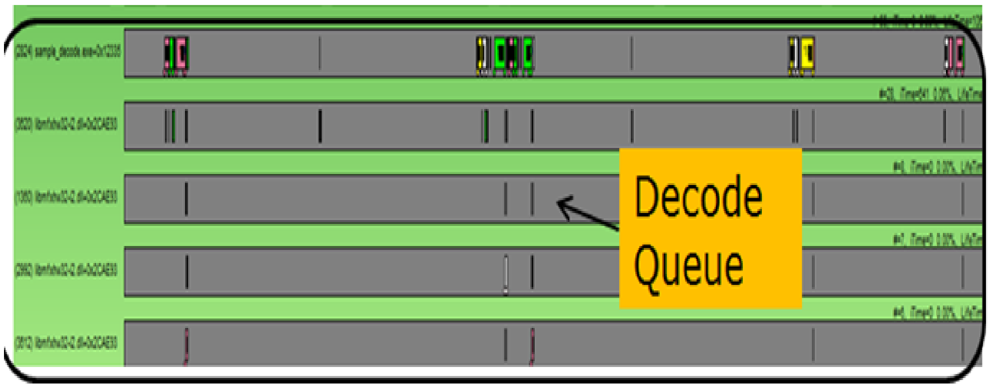


Figure 7.10: Impact of GPU offload on CPU.

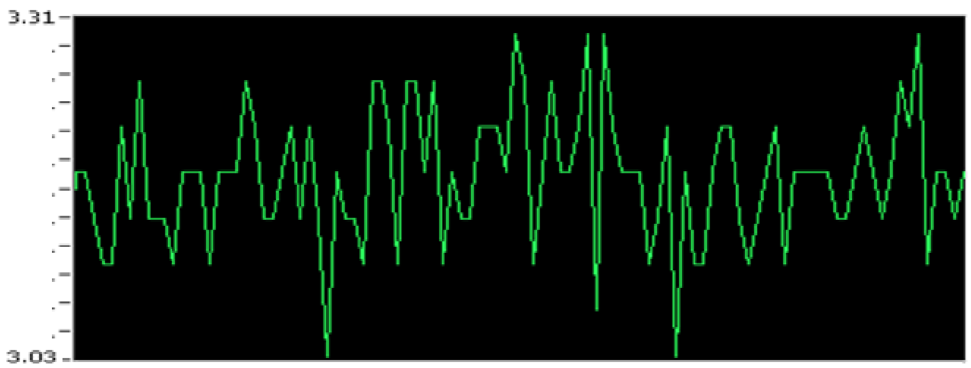


Figure 7.11: Power savings real-time for GPU offload.

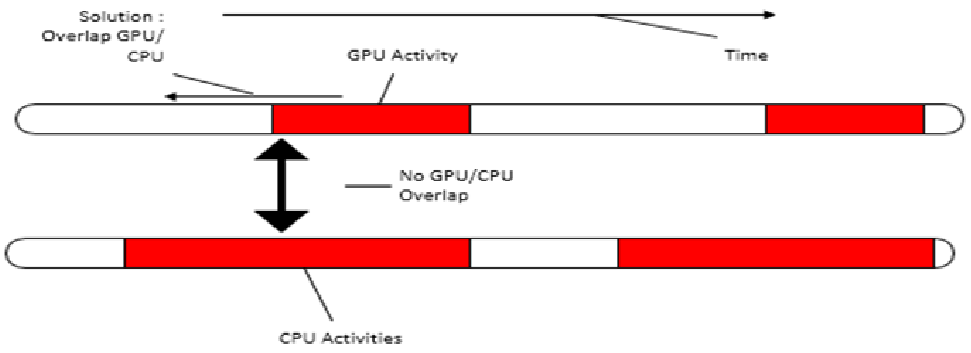


Figure 7.12: CPU/GPU concurrency and overlap.

platforms, optimization is more important. Power estimation of SoC apart from the platform is difficult. In this case study, we provide analysis of audio offload and power estimation using DAQ.

We used Microsoft hardware offload for audio processing to find the impact of audio offload on and off power. In addition, we used Blender foundation Elephant dreams for testing the audio and its optimization [41]. Using Windows 8 audio architecture, we created a knob in the media playback application to use audio DSP.

As shown in Figure 7.13, two scenarios were tested for the requirements of fine-grained IP units: Audio DSP by bypassing the SW audio engine, and another by directly doing HW offload [52].

We did experiments on audio playback to verify the impact of power on IP blocks. Figure 7.16 shows the power delta savings in watts when using audio IP unit offload compared to without offload. The approximate power savings observed by playing a 48-KHz audio file was 350 mW. It is significant since it can give between 10 and 15 minutes more listening during playback.

Most of the savings happens due to dedicated processing of the work. IP block helped to reduce the CPU's wake-ups and continued processing the audio stream without wake-up. Figure 7.15 shows CPU wake-up sampled at every 1 msec using Event tracing for Windows logs. The red line represents wake-ups without offloading, while the blue shows when audio offload is enabled. The impact of power during

	Software-Based Estimation	DAQ-Based Estimation
CPU $\rightarrow$ GPU Offload Savings	4.1	4.19
GPU $\rightarrow$ CPU Concurrency	0.08	0.239
GPU $\rightarrow$ CPU Overlapping	0.01	0.129

Table 7.1: Estimation of power variance using software- and hardware-based tools.

continuous playing without a pre-buffer shows offloading to a dedicated IP unit can help the CPU to go into a deep-power sleep state and even power off completely until the IP unit complete the processing.

7.6 Analysis and Conclusion

In this chapter, we presented studies on using SoC and platform IP offload. This chapter shows the need for low-power sophisticated measurement tools to understand the impact of a low-power IP unit available on the platform. We also showed that using only an IP unit will not result in optimal power optimization; therefore, there is still a need for optimization that can be achieved by proper alignment of CPU activities with IP block activities. Optimization can only be captured by DAQ instrumentation instead of software-based calculation. Compared to software-based solutions for low-power measurement, DAQ provides accuracy and less variance. Results show 12% error as we move to low-power SoC when we use software-based measurements compared to the DAQ because current software-measurement tools can miss a small IP unit.

As future work, we plan to investigate how to incorporate low-cost, power-friendly techniques such as buffering the hardware counter before it flushes the data to disk. We will also investigate automatic energy-saving measures on IPs on the platform using low-cost overhead software counters or DAQ.

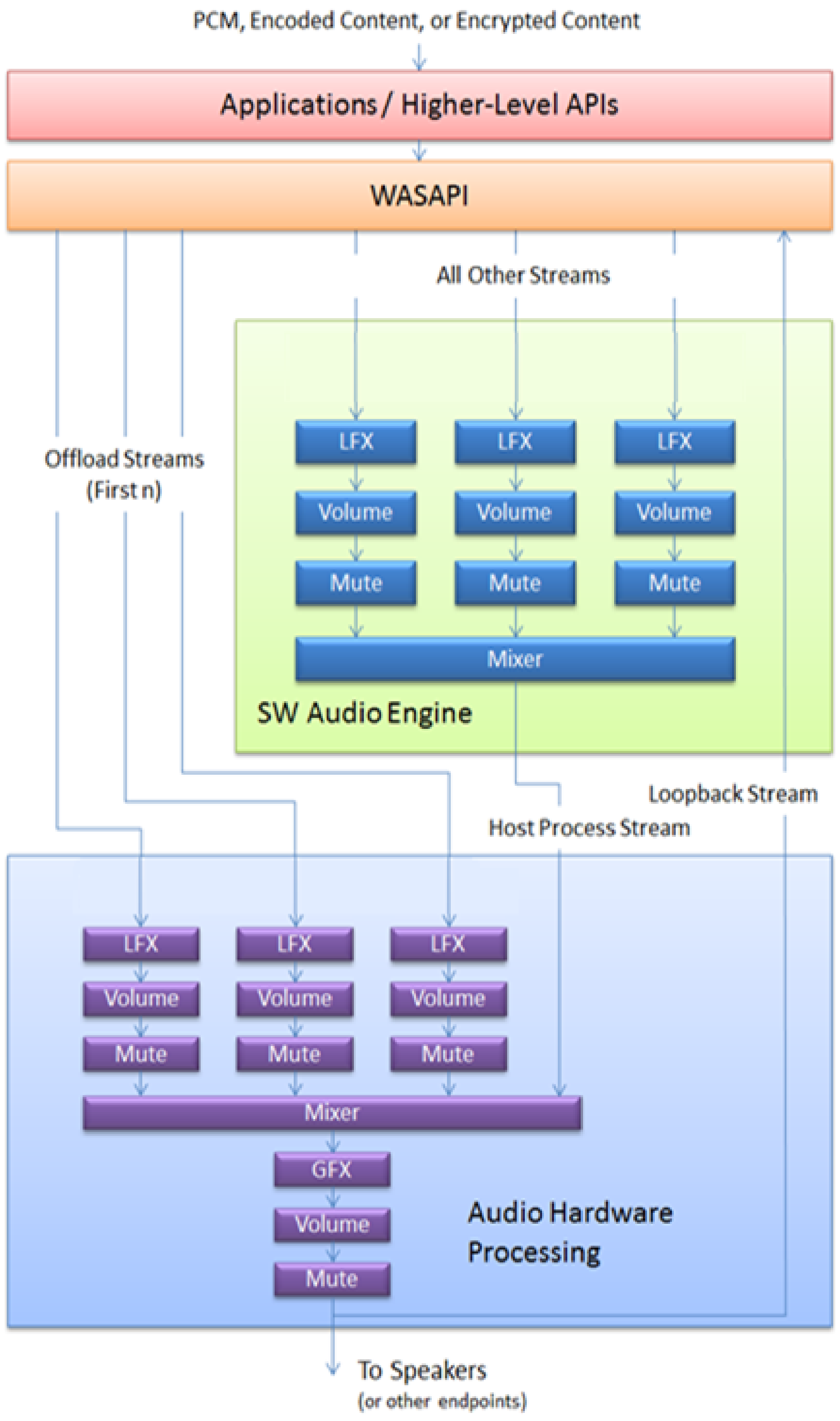

PCM, Encoded Content, or Encrypted Content
Applications / Higher-Level APIs
WASAPI
All Other Streams
Offload Streams (First n)
LFX
Volume
Mute
Mixer
SW Audio Engine
Loopback Stream
Host Process Stream
GFX
Audio Hardware Processing
To Speakers (or other endpoints)

Figure 7.13: Microsoft audio block diagram.

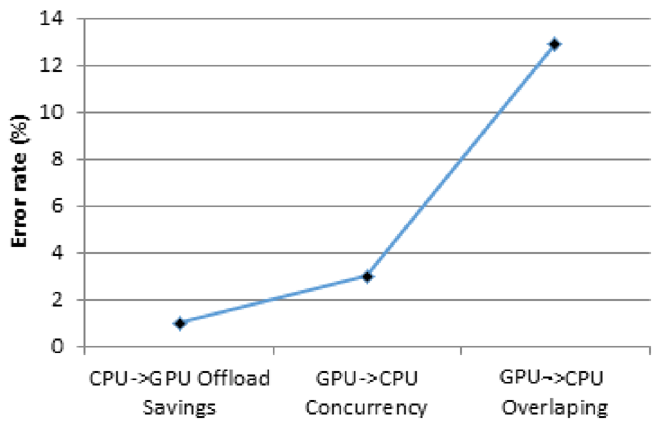


Figure 7.14: Error graph on low-power SoC savings.

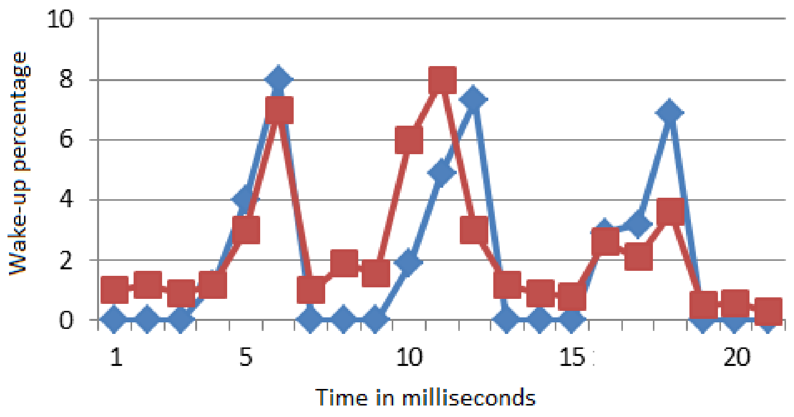


Figure 7.15: Overtime view for audio activities.

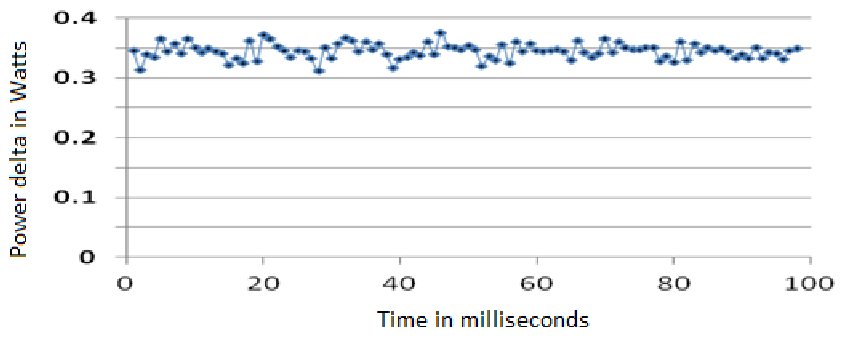


Figure 7.16: Delta of power savings achieved by offloading.

CHAPTER 8: CONCLUSION AND FUTURE WORK

In this Ph.D. dissertation, we presented our research accomplishments in the field of energy efficiency of mobile devices. We presented energy overhead analysis of mobile devices and a survey of relevant literature to lay the foundation of our work. We focused in this dissertation on the energy-efficiency analysis of mobile devices. Through our analysis, we were able to identify several issues that contributed to the energy inefficiencies of mobile devices and proposed optimization techniques. In addition, our analysis enabled us to provide application development techniques which can increase the energy efficiency of mobile apps. Moreover, we developed three tools. The first tool is *SoftPowerMon*, which can power profile Android platforms in order to expose the power consumption behavior of the CPU. The second tool is *EnergyMeter*, which can collect the energy consumption of Windows platforms, in addition to the energy consumption of package, cores, and GPU of HASWELL ULT chipset. The third tool is *BatteryExtender*, an adaptive user-guided tool for power management of mobile devices. The tool enables users to extend battery life on demand for a specific duration until a particular task is performed. Finally, we examined the power consumption of Systems-on-Chips (SoCs) and observed the impact of offloading tasks, from the CPU to the specialized custom engines, on the energy efficiency. Based on our case studies, we were able to showcase that current software-based power profiling techniques for SoCs can have an error rate close to 12%.

Even though our contribution to increase the energy efficiency of mobile devices is significant, there is still opportunity for further optimization of the energy efficiency of mobile devices. In particular, we aim to pursue the following research directions in the future:

- We are planning on improving our *BatteryExtender* tool by improving our resource power-consumption estimation by continuously profiling the platform when battery life is not an important resource for the user.

- Battery life is a significantly important resource and users tend to evaluate devices based on their expected battery life. However, battery consumption is not strictly a hardware issue, but an app issue as well. Yet, there are no tools that can rank apps based on their energy efficiency as it is common to rank hardware devices based on their energy efficiency. As a result, we are planning to develop a tool which can automatically energy profile apps and rank them in terms of their energy efficiency.

- We are also planning on developing a software-based tool to power profile the SoC devices with very low error rate. Then, using our profiling tool, we can identify further energy inefficiency causes and attempt to solve them.

- Wearable technology is becoming the hot new technology. This too has a very limited power sources. As a result, we are planning to use the knowledge gained from developing power profiling tools for the SoC with high precision; then we can use that knowledge to create tools specifically for wearable technology.

- We exposed in our case studies the energy inefficiency of mobile web application compared to native apps. However, since creating native apps for multiple operating systems can be costly for organizations, we are planning to find optimization techniques in order to reduce the gap between the energy consumption of native versus web apps.

REFERENCES

[1] https://developer.apple.com/library/mac/##documentation/developertools/conceptual/InstrumentsUserGuide/Introduction/Introduction.html.

[2] http://code.google.com/p/iptableslog/.

[3] http://www.3c71.com/android/?q=node/1##main-content-area.

[4] http://android.nextapp.com/site/systempanel.

[5] 2013 roundup of smartphone and tablet forecasts and market estimates. http://www.forbes.com/sites/louiscolumbus/2013/01/17/2013-roundup-of-mobility-forecasts-and-market-estimates/.

[6] 2640a/2645a netdaq networked data acquisition unit user manual. http://www.testequipmentdepot.com/fluke/pdf/netdaq_manu.pdf.

[7] About instruments. https://developer.apple.com/library/mac/documentation/developertools/conceptual/instrumentsuserguide/Introduction/Introduction.html.

[8] About the ios technologies. https://developer.apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview/Introduction/Introduction.html.

[9] Amazon unbox vido player. http://www.amazon.com/gp/video/ontv/player.

[10] Device power management. http://msdn.microsoft.com/en-us/library/windows/hardware/dn495664(v=vs.85).aspx.

[11] Event tracing. http://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx.

[12] Google v8 benchmark suite. http://v8.googlecode.com/svn/data/benchmarks/v5/run.html.

[13] Guidelines for tiles and badges (windows store apps). http://msdn.microsoft.com/en-us/library/windows/apps/hh465403.aspx.

[14] Instruments users guide. http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html.

[15] Intel 64 and ia-32 architectures software developers manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[16] Oprofile. http://oprofile.sourceforge.net/about/.

[17] Power efficient multimedia play back on mobile platforms", author=Agrawal, A and Huff, T and Potluri, S and Cheung, A, and Thaku, A and Holland, H, booktitle="intel tech nology journal", volume=15, issue=2, year=2011.

[18] Push notification overview (windows store apps). http://msdn.microsoft.com/en-us/library/windows/apps/hh913756.aspx.

[19] Sunspider javascript benchmark. http://www.webkit.org/perf/sunspider/sunspider.html.

[20] Trepn profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

[21] Why apple's 2014 won't be like 2013. http://news.cnet.com/8301-13579_3-57615293-37/why-apples-2014-wont-be-like-2013/.

[22] Windows performance analyzer. `http://msdn.microsoft.com/en-us/library/windows/hardware/hh448170.aspx`.

[23] Windows performance recorder. `http://msdn.microsoft.com/en-us/library/windows/hardware/hh448205.aspx`.

[24] Worldwide smartphone markets: 2011 to 2015. `http://www.researchandmarkets.com/research/7a1189/worldwide\_smart`, 2011.

[25] Nena Innovation AB. Nenamark2. `http://nena.se/nenamark/view?version=2`.

[26] Ahmed Abdelmotalib and Zhibo Wu. Power consumption in smartphones (hardware behaviourism). *International Journal of Computer Science Issues (IJCSI)*, 2012.

[27] Kevin Ashton. That internet of things thing. *RFiD Journal*, 22:97–114, 2009.

[28] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.

[29] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 37–42. ACM, 2000.

[30] Fehmi Ben Abdesslem, Andrew Phillips, and Tristan Henderson. Less is more: energy-efficient mobile sensing with senseless. In *Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, pages 61–62. ACM, 2009.

[31] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. volume 28, pages 83–94. ACM, 2000.

[32] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[33] ARM Info Center. Cortex a-9 technical reference manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex\_a9\_r4p1\_trm.pdf`.

[34] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[35] Sunny Consolvo, David W McDonald, Tammy Toscos, Mike Y Chen, Jon Froehlich, Beverly Harrison, Predrag Klasnja, Anthony LaMarca, Louis LeGrand, Ryan Libby, et al. Activity sensing in the wild: a field trial of ubifit garden. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1797–1806. ACM, 2008.

[36] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53. IEEE, 2012.

[37] Android developers. Android debug bridge. `http://developer.android.com/tools/help/adb.html`.

[38] Thanh Do, Suhib Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool, 2009.

[39] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. Power modeling of graphical user interfaces on oled displays. In *Proceedings of the 46th Annual Design Automation Conference*, pages 652–657. ACM, 2009.

[40] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348. ACM, 2011.

[41] Elephant Dreams. Elephant dreams. `http://www.elephantsdream.org/`, 2013.

[42] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M Blackburn, and Kathryn S McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling, 2011.

[43] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 281–287. ACM, 2010.

[44] Hossein Falaki, Ratul Mahajan, and Deborah Estrin. Systemsens: a tool for monitoring usage in smartphone research deployments. In *Proceedings of the sixth international workshop on MobiArch*, pages 25–30. ACM, 2011.

[45] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.

[46] Jason Flinn and Mahadev Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications, 1999.

[47] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. Comet: code offload by migrating execution transparently. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, OSDI*, volume 12, pages 93–106, 2012.

[48] Tor-Morten Grønli, Jarle Hansen, and Gheorghita Ghinea. Android vs windows mobile vs java me: a comparative study of mobile development environments. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, page 45. ACM, 2010.

[49] Tor-Morten Grønli, Jarle Hansen, and Gheorghita Ghinea. A cloud on the horizon: the challenge of developing applications for android and iphone. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, page 64. ACM, 2011.

[50] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, Narayanan Vijaykrishnan, and Mahmut Kandemir. Using complete machine simulation for software power estimation: The softwatt approach, 2002.

[51] Hao Han, Yunxin Liu, Guobin Shen, Yongguang Zhang, and Qun Li. Dozyap: power-efficient wi-fi tethering. In *Proceedings of the 10th international conference on mobile systems, applications, and services*, pages 421–434. ACM, 2012.

[52] Windows Dev Center Hardware. Hardware offload of audio processing test (system). `http://msdn.microsoft.com/en-us/library/windows/hardware/hh997936.aspx/`, 2013.

[53] Windows Dev Center Hardware. Using gpuview. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff570133(v=vs.85).aspx/`, 2013.

[54] Gael Hofemeier. Ultrabook and tablet windows* 8 sensors development guide. `http://software.intel.com/en-us/articles/ultrabook-and-tablet-windows-8-sensors-development-guide`, 2013.

[55] Nick Honetschlager. Mobile applications in android.

[56] Kishonti Informatics. Glbenchmark. `http://gfxbench.com/result.jsp`.

[57] Texas Instrumentation. Data acquisition with pxi and pxi express. `http://www.ni.com/data-acquisition/pxi/`, 2013.

[58] Intel. 4th generation intel core i5 processor. `http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html`, 2013.

[59] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. Devscope: a nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 353–362. ACM, 2012.

[60] J.Y. Kang, M.J. Park, C. Lee, and S.G. OH. User interface method and apparatus therefor, January 9 2014. US Patent App. 13/928,919.

[61] Aman Kansal, Scott Saponas, AJ Brush, Kathryn S McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 661–676. ACM, 2013.

[62] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design, 2008.

[63] Kyu-Han Kim, Alexander W Min, Dhruv Gupta, Prasant Mohapatra, and Jatinder Pal Singh. Improving energy efficiency of wi-fi sensing on smartphones. In *INFOCOM, 2011 Proceedings IEEE*, pages 2930–2938. IEEE, 2011.

[64] Marcello Lajolo, Anand Raghunathan, Sujit Dey, and Luciano Lavagno. Efficient power co-estimation techniques for system-on-chip design. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 27–34. IEEE, 2000.

[65] Marcello Lajolo, Anand Raghunathan, Sujit Dey, and Luciano Lavagno. Cosimulation-based power estimation for system-on-chip design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(3):253–266, 2002.

[66] Nicholas D Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T Campbell. A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140–150, 2010.

[67] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. 2013.

[68] Frank Maker, Rajaveen Amirtharajah, and Venkatesh Akella. Update rate tradeoffs for improving online power modeling in smartphones. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 114–119. IEEE, 2013.

[69] Dustin McIntire, Thanos Stathopoulos, and William Kaiser. etop-sensor network application energy profiling on the leap2 platform, 2007.

[70] G. Metri, M. Sabharwal, S. Iyer, and A. Agrawal. Hardware/software codesign to optimize soc device battery life. *Computer*, 46(10):89–92, October 2013.

[71] Grace Metri, Abhishek Agrawal, Ramesh Peri, Monica Brockmeyer, and Weisong Shi. A simplistic way for power profiling of mobile devices. In *Energy Aware Computing, 2012 International Conference on*, pages 1–6, Dec 2012.

[72] Grace Metri, Abhishek Agrawal, Ramesh Peri, and Weisong Shi. What is eating up battery life on my smartphone: A case study. In *Energy Aware Computing, 2012 International Conference on*, pages 1–6. IEEE, 2012.

[73] Antti P Miettinen and Jukka K Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 4–4. USENIX Association, 2010.

[74] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 317–328. ACM, 2012.

[75] M Motlhabi. Advanced android power management and implementation of wakelocks. *University of the Western Cape. Paper available online: http://www. cs. uwc. ac. za/˜ mmotlhabi/apm2. pdf (nd)*, 2008.

[76] Adrian Mullally, Nigel McKelvey, and Kevin Curran. Performance comparison of enterprise applications on mobile operating systems. *Telkomnika*, 9(3), 2011.

[77] Min Mun, Sasank Reddy, Katie Shilton, Nathan Yau, Jeff Burke, Deborah Estrin, Mark Hansen, Eric Howard, Ruth West, and Péter Boda. Peir, the personal environmental impact report, as a platform for participatory sensing systems research. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 55–68. ACM, 2009.

[78] Adam Oliner, Anand Padmanabha Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. 2013.

[79] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[80] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM, 2011.

[81] Gian Paolo Perrucci, Frank HP Fitzek, and Jörg Widmer. Survey on energy consumption entities on the smartphone platform. In *Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd*, pages 1–6. IEEE, 2011.

[82] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Characterizing radio resource allocation for 3g networks. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 137–150. ACM, 2010.

[83] Freeman Rawson and IBM Austin. Mempower: A simple memory power analysis tool set. *IBM Austin Research Laboratory*, 2004.

[84] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A comparison of high-level full-system power models. *HotPower*, 8:3–3, 2008.

[85] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge, 2012.

[86] M. Sabharwal, A. Agrawal, and G. Metri. Enabling green it through energy-aware software. *IT Professional*, 15(1):19–27, 2013.

[87] M. Sabharwal, G. Metri, Chao Huang, and A. Agrawal. Towards fine grain power profiling tools for soc based mobile devices. In *Energy Aware Computing Systems and Applications (ICEAC), 2013 4th Annual International Conference on*, pages 87–92, Dec 2013.

[88] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures, 2009.

[89] Alex Shye, Benjamin Scholbrock, Gokhan Memik, and Peter A Dinda. Characterizing and modeling user activity on smartphones: summary. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 375–376. ACM, 2010.

[90] Suresh Siddha, Venkatesh Pallipadi, and AVD Ven. Getting maximum mileage out of tickless. In *Linux Symposium*, volume 2, pages 201–207. Citeseer, 2007.

[91] Intel Software. Intel media sdk 2013. `http://software.intel.com/en-us/vcsource/tools/media-sdk`, 2013.

[92] Charles Sterling. Energy consumption tool in visual studio 2013. `http://blogs.msdn.com/b/visualstudioalm/archive/2013/07/10/energy-consumption-tool-in-visual-studio-2013.aspx`, 2013.

[93] Sasu Tarkoma and Eemil Lagerspetz. Arching over the mobile chasm: Platforms and runtimes. *Computer*, 2010.

[94] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 85–98. ACM, 2009.

[95] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who killed my battery?: analyzing mobile browser energy consumption. In *Proceedings of the 21st international conference on World Wide Web*, pages 41–50. ACM, 2012.

[96] Olivier Tremblay and Louis-A Dessaint. Experimental validation of a battery dynamic model for ev applications. *World Electric Vehicle Journal*, 3:13–16, 2009.

[97] Narseo Vallina-Rodriguez, Pan Hui, Jon Crowcroft, and Andrew Rice. Exhausting battery statistics: understanding the energy demands on mobile handsets. In *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, pages 9–14. ACM, 2010.

[98] Narayanan Vijaykrishnan, M Kandemir, Mary Jane Irwin, Hyun Suk Kim, and Wu Ye. Energy-driven integrated hardware-software optimizations using simplepower, 2000.

[99] Chengke Wang, Fengrun Yan, Yao Guo, and Xiangqun Chen. Power estimation for mobile applications with profile-driven battery traces. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 120–125. IEEE, 2013.

[100] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 213–220. ACM, 2013.

[101] Z Yang. Powertutor–a power monitor for android-based mobile platforms. *EECS, University of Michigan, retrieved September*, 2, 2012.

[102] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Y Wang. Modeling hard-disk power consumption. In *FAST*, volume 3, pages 217–230, 2003.

[103] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones, 2010.

ABSTRACT

ENERGY-EFFICIENCY ANALYSIS AND OPTIMIZATION FOR MOBILE PLATFORMS

by

GRACE CAMILLE METRI

August 2014

Co-Advisor: Dr. Monica Brockmeyer

Co-Advisor: Dr. Weisong Shi

Major: Computer Science

Degree: Doctor of Philosophy

The introduction of mobile devices changed the landscape of computing. Gradually, these devices are replacing traditional personal computer (PCs) to become the devices of choice for entertainment, connectivity, and productivity. There are currently at least 45.5 million people in the United States who own a mobile device, and that number is expected to increase to 1.5 billion by 2015.

Users of mobile devices expect and mandate that their mobile devices have maximized performance while consuming minimal possible power. However, due to the battery size constraints, the amount of energy stored in these devices is limited and is only growing by 5% annually. As a result, we focused in this dissertation on energy efficiency analysis and optimization for mobile platforms. We specifically developed *SoftPowerMon*, a tool that can power profile Android platforms in order to expose the power consumption behavior of the CPU. We also performed an extensive set of case studies in order to determine energy inefficiencies of mobile applications. Through our case studies, we were able to propose optimization techniques in order to increase the energy efficiency of mobile devices and proposed guidelines for energy-efficient app development. In addition, we developed *BatteryExtender*, an adaptive user-guided tool for power management of mobile devices. The tool enables users to extend battery

life on demand for a specific duration until a particular task is completed. Moreover, we examined the power consumption of System-on-Chips (SoCs) and observed the impact on the energy efficiency in the event of offloading tasks from the CPU to the specialized custom engines. Based on our case studies, we were able to demonstrate that current software-based power profiling techniques for SoCs can have an error rate close to 12%, which needs to be addressed in order to be able to optimize the energy consumption of the SoC. Finally, we summarize our contributions and outline possible direction for future research in this field.

AUTOBIOGRAPHICAL STATEMENT

Grace Metri is a Ph.D. candidate at the Department of Computer Science at Wayne State University where she received her M.S. in Computer Science in 2010 and B.S. in Computer Science with a minor in Business in 2006. Her research interests are power and energy profiling tools for mobile devices, power management in data centers, energy efficient software, and increasing energy efficiency of data centers and mobile devices. She published several papers in different venues, including IEEE Computer Magazine, IT Pro Magazine, Energy Aware Computing Systems and Applications (ICEAC), and IEEE Cloud. Her work experience include both academia and industry. She was a Graduate Teaching Assistant for several years and worked as a lecturer for one academic year. She also worked as a Software Engineer at Thomson Reuters before joining Intel to work on research related to power and energy profiling of mobile devices.