BatteryExtender: An Adaptive User-Guided Tool for Power Management of Mobile Devices

Grace Metri[1], Weisong Shi[1], Monica Brockmeyer[1], and Abhishek Agrawal[2]
[1]Department of Computer Science, Wayne State University, Detroit, Michigan
[2]Software and Services Group, Intel Corporation, Santa Clara, California
[1]{gmetri, weisong, mbrockmeyer}@wayne.edu and [2]abhishek.r.agrawal@intel.com

ABSTRACT

The battery life of mobile devices is one of their most important resources. Much of the literature focuses on accurately profiling the power consumption of device components or enabling application developers to develop energy-efficient applications through fine-grained power profiling. However, there is a lack of tools to enable users to extend battery life on demand. What can users do if they need their device to last for a specific duration in order to perform a specific task? To this extent, we developed BatteryExtender, a user-guided power management tool that enables the reconfiguration of the device's resources based on the workload requirement, similar to the principle of creating virtual machines in the cloud. It predicts 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 decreased the energy consumption 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%.

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. Many factors can impact battery life. Resource utilization by applications running on the platform and the number of powered-on device components the platform can greatly impact battery life. As a result, the platform's power management layer 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.

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 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. It 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.

To this extent, we discuss related work and our motivation. Then, we present background information followed by BatteryExtender design and implementation. Then, we present our evaluating followed by our conclusion and Future Work.

RELATED WORK

Increasing the battery life of mobile devices has been heavily investigated by researchers. In order to reach this goal, researchers have taken four different approaches.

Power-profiling models of hardware components of mobile devices. Using external power-measurement tools, Carroll and Heiser [4] analyzed the power consumption of smartphone components using a Data Acquisition system (DAQ). Based on their analysis, the display, GSM module, graphics accelerator/driver, and backlight were the most power-consuming components. Dong *et al.* [6] also relied on external measurement tools in order to power profile the graphical user interface on OLED displays. By only relying on software-based techniques, both Maker *et al.* [12] and Jung *et al.* profiled the power consumption of smartphone components using the battery management unit (BMU).

Estimating energy consumption of applications to enable developers develop energy-efficient applications. WattsOn [14] allows developers 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 the smartphone manufacturer or OS platform developers. Likewise,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Ubicomp '14, Spetember 13–17, 2014, Seattle, WA, USA
Copyright © 2014 978-1-4503-2968-2/14/09$15.00.
http://dx.doi.org/10.1145/2632048.2632082

Eprof's [17] captures and accounts for the power usage of the program entity in order to find the source code of energy bugs. In addition, Pathak *et al.* [18] provide fine-grained power modeling for smartphones using system call tracing which account for components' power based on their state and their tail power, and then associate the values with the application responsible for the power consumption.

Providing APIs for developers to either increase their applications energy efficiency or power profile it. Senergy was developed by Kansal *et al.* [10]. 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. Another framework example is SystemSens [7], 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. Application developers can use the interface as a virtual sensor of the framework and thus collect context and power utilization data related to the application.

Providing users with power-profiling tools that highlight the impact of running applications on the platform. Most of the tools in this category rely on collective information to build the energy consumption models. For instance, Carat [16] is a tool that sends coarse-grained statistics to servers residing in the cloud. 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 energy bugs.

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 and utilization counters.

MOTIVATION

Our motivation stems from two facts: The first is the lack of research/tools that enable users to extend battery life on demand as we have shown in 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 and iOS devices, such as *Battery Booster, Battery Defender, Battery Dr. Saver, Battery Extender, Battery Indicator, Battery Info, Battery Monitor Pro, Top Battery, Battery Doctor, Battery Expert, Battery Life Pro, Battery Magic Elite, 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 *"Video Playback time,"* however, their recommendations are general and not specific for a given app. As a result, because apps of the same category can each consume vastly different amounts of energy, the apps' recommendations can be completely off and not very useful in many cases. Finally, to the best of our knowledge, we did not find a tool that can answer **"what can users do if they need to extend battery life in order to accomplish a specific task?"**

The second fact is 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 such as 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 display brightness based on the device's surrounding light exposure. Another example is their usability as a mean for collaborative diagnostics, such as the case of Carat [16], which collects information from its user base to perform energy diagnostics of applications. Another example is UbiFit Garden [5], which uses mobile sensors to capture physical activities of its users and associate the information with their physical goals. PEIR [15] is another project that uses sensors in mobile devices to alert users about their carbon footprint. Lane *et al.* [11] provide a comprehensive survey of current mobile phone sensing projects. 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 many new sophisticated sensors will be introduced causing an increase in power consumption of the platform, 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 (camera, Wi-Fi, battery, sensors, etc) , 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.

BACKGROUND: ENERGY OVERHEAD ANALYSIS

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

Application Layer: Applications running on a platform increase the power consumption based on their resource utilization. Rivoire *et al.* [19] 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.

Management Layer: The management layer manages 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 changes 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. 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.

Hardware Layer: The number of components in 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. In this paper, we are interested in a detailed discussion of network devices, sensors, utility devices, and display.

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 [4]) as the most power-consuming device in this category. 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 [13], 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.

Sensors: Mobile devices are built with 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 [8]. 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 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 [9] transformed the usability of a camera from a utility device, which strictly records videos or takes snapshots, to a sensing device. The patent states that the technology allows to use the camera to acquire images, divide them into 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: 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. The refresh rate, which can be measured in hertz, is 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. Finally, both brightness and refresh rate need to be considered when evaluating the power consumption of displays.

Case Study: Windows Device Power-Management Analysis

Starting with Windows 8, Microsoft requires that the device components of their platform support five different D-States [2]. The first state is D0, which is the active state, followed by D1, the highest-powered sleep state, where the device preserves it hardware context. Then, D2 where most context is lost by the hardware. The fourth state is D3-Hot, where the device draw very low current but can be detected by the bus. The last state is D3-Cold, the lowest possible sleep state. After a device enters D3-Cold for a period of time, it gets turned off. In order to validate our approach we performed preliminary study to determine the actual device power sleep states during two scenarios. The first scenario is the platform default state, 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 idle with the screen on for both scenarios. We compared the list of devices collected between default settings and reconfigured settings. We noted that greater than 50 devices were detected using the default scenario compared to the disabled one as shown in Table 1. 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. Many devices were in D0 state 100% of the time, and just a few were in D0 for 96.9% of the time before they entered D2 for the remaining duration of 3.10%. 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. The purpose is to determine, over a long duration, how the device states change if the devices are not been actively used. The results are displayed in Table 1 in the video playback section. Again, many devices remained in active state 100% 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% 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 during Idle scenario. Finally, this experiment demonstrated that a better power-management mechanism was needed.

BATTERYEXTENDER DESIGN

The BatteryExtender (BE) tool major goal is to enable extending battery life on demand in the event that battery life resource has high importance. As a result, we set the following conditions:

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	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 1. 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.

- **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.

- **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 satisfy their needs.

Extending Battery Life Feature Design

Based on our goals, we define extending battery life design as shown in Figure 1. It consists of five components: calibration, user interactive (user command selection and user interface), energy profiling, power management, and monitoring modules. The calibration module aims to profile the power consumption of platform device components and save it to a configuration file. 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 that determines the list of applications running on the platform and calculates the estimate of their minimal energy consumption over the battery life and ranks the top 5 most energy-consuming applications. It also determines each device component power state (on or off) and active state (e.g, Bluetooth is actively connected to a device and transmitting or receiving data). Then, it estimates the impact of changing the device component power state on the platform's battery life. Upon completion, the energy profiling module updates the user interface with the top 5 power-consuming applications and displays the current active state of the device components in ad-

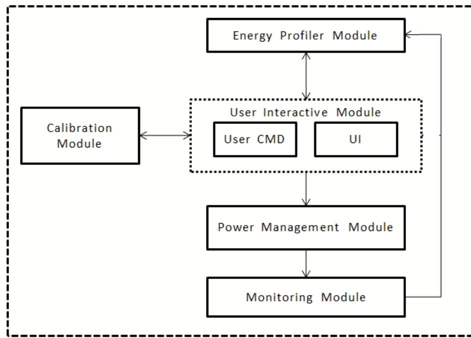

Figure 1. Extending battery life design.

dition to displaying the amount of battery life saved/gained by changing their power state. At this point, the user can select the options to change. Upon option selection confirmation, the power management module re-configures the device 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, the energy profiling module is triggered again. Then, it is up to the user to either reconfigure the platform or accept the new expected battery life.

Additional Feature: Live Monitoring

Even though BatteryExtender tool main feature is extending battery life on demand, we created another useful feature for *live monitoring*. We created this feature because power consumption of a platform is not consistent. Therefore, if users want to simply monitor the changes in platform power consumption, they can use this feature. There are two types of live monitoring as follows:

Battery Usage Interface: It enables users to collect a log of battery metrics, including timestamp, capacity, discharge rate, voltage, and expected battery life. Users can save the collected log to a csv file or delete the data and start over.

Application Usage Interface: It continuously profiles running applications (similar to energy profiling technique for extending battery life with the exception that profiling in this case is continuous) and provides the following metrics per application: the minimum power consumption, and the average minimum power consumption since profiling began.

Discussion: BatteryExtender Usability by Novice Users

BatteryExtender concept is receiving significant interest from mobile users. Since the tool main audience target is end-users from all levels of expertise, therefore, its sucess is definitely based on its ease of usability and intuitive design.

In order to achieve this goal, we adopted few techniques which we predict to enable novice users to adopt our tool. First of all, all the device components which may be disabled/enabled by users were selected carefully to avoid unpredictable/disabling impact on the platform itself. For instance, disabling HD is not included as one of the possible device components to disable/enable by users even though its implementation is straight forward. Second, when the users are given the impact of disabling/enabling device components, they are given the status of the device (active/inactive). We hope this feature may be used as an indicator to users about whether the component is actively needed or not. This solution is not a guarantee that the 'disabled' component may not be needed in the duration of the battery extension; however, from our experience with popular apps, if a component was needed but disabled/unavailable, the apps tend to notify the users about it. Nonetheless, we added an option for users to "reset the device configuration" to the configuration prior to extending battery life. As a result, if the user disabled a needed component which hindered an app usability—and lacked the technical knowledge—the user can easily reset the platform. Third, upon battery extension expiration, BatteryExtender "resets the device configuration" even if battery was drained prior to completing the task (it resets the platform upon the next device power on.)

Finally, we hope the measures we adopted ensure the usability of the tool even by novice users but we expect to evolve and improve the usability as we get feedback from future field studies.

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 platform contains an extensive number of components, and our power-management approach can significantly impact the battery life. During our experiments, we used two different Windows platforms. The first platform is a Dell XPS 12 Ultrabook Convertible, as described in Table 2, 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." The second platform is the Microsoft Surface 2 Pro Tablet, as described in Table 3. In the remainder of this paper, we will refer to this platform as "Surface."

Preliminary Experiments: Analysis of Collection Granularity of Battery Life

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. We are interested in

Specification	Description
Platform	Dell Ultrabook Convertible, Windows 8 Pro
Processor	Intel(R) Core(TM) i7-4650U Code Name: HASWELL-ULT
Hard Disk	256 GB Solid State
Memory	8.0 GB
Display	12.5" Full HD, refresh rate 59 and 60 Hz
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
Camera	Front WebCam
Sensors	HID Sensor Collection Simple Device Orientation Sensor Microsoft VS Location Simulation Sensor

Table 2. Dell Ultrabook Convertible specifications.

Specification	Description
Platform	Microsoft Surface 2 Pro, Windows 8.1 Pro
Processor	Intel(R) Core(TM) i5-4200U Code Name: HASWELL-ULT
Hard Disk	64 GB Solid State
Memory	4.0 GB
Display	10.6" HD, refresh rate 60 Hz
Bluetooth	Marvell AVASTAR Bluetooth Radio Adapter
Wi-Fi	Marvell AVASTAR 350N
Speaker & Microphone	Realtek High Definition Audio
Touch	10 Touch Points and Pen
Camera	Microsoft LifeCam Front and Rear
Sensors	HID Sensor Collection Simple Device Orientation Sensor Microsoft VS Location Simulation Sensor

Table 3. Microsoft Surface 2 Pro Tablet specifications.

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. Rate reported in milliwatts (mW) and denoted as B_{Rate}. It is the amount of power drawn from the battery. Battery life remaining, denoted as B_{Life} is the fraction of B_{Cap} over B_{Rate}.

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 high overhead. As a result, we conducted preliminary studies in order to find an optimal coarse grained interval of battery information collection. Profiling the battery life behavior consists of the following steps. First, we disabled all power-management functionality of the platform power plan and all network adapters in order to maintain consistent power consumption of the platform. Then, we disconnected the power cable of a fully charged battery. Then, we kept the platform in idle mode with the screen on. Finally, we let the battery drain while collecting, at a time interval T, the remaining capacity and battery life remaining in seconds. Figure 2 represents the data collected on a Surface at a 3-second interval, and Figure 3 represents the data collected on Dell 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 (3-second

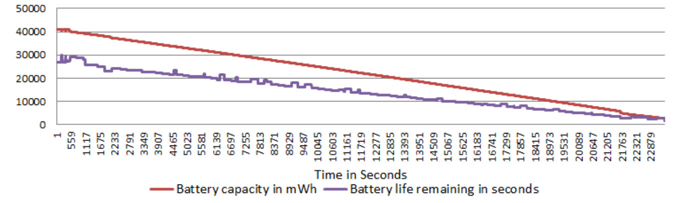

Figure 2. Relationship between battery capacity and remaining battery life over time at a 3-second interval on Surface 2 Pro tablet.

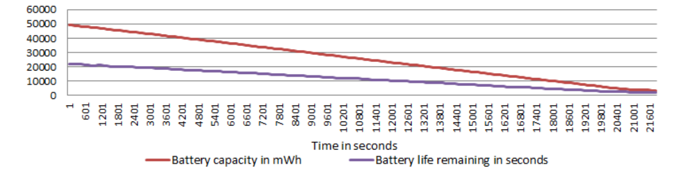

Figure 3. Relationship between battery capacity and remaining battery life over time at a 2-minute interval on Dell convertible.

interval), resulted in an unclear picture because of high variance. On the other hand, collecting at a relatively high granularity (a 2-minute interval), let us collect the data with low variance, which let us be more accurate.

Implementation

The implementation of the five components of extending battery life feature are as follows:

Calibration Module

Power consumption of a platform is highly dependent on the component collection it contains. Since components vary from one platform to another, and even components from different vendors can vary their power consumption, it becomes important to estimate the their power consumption through a self-modeling approach. Once BatteryExtender is installed, users are required to run calibration at least once but they may choose to run it periodically in order to improve accuracy of BatteryExtender due to the fact that the battery's resistance changes over time.

Most 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 disable both prior to calibration in order to keep the platform's power consumption constant during the calibration phase. Then, the calibration module automatically performs the following steps:

1. Terminate all running applications with the exception of BatteryExtender.
2. Get the list of all *Plug and Play devices* and disable all devices which a user can sacrifice when battery life is needed.
3. Set the display brightness to 75% because we noticed when platforms are on battery, the power-management module changes the display brightness to 75.
4. Get the current display refresh rate.
5. Sleep for 120 seconds to avoid overhead from our changes.

6. 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 1.
7. Select a device from the list of PnP devices and enable it.
8. Sleep for 120 seconds.
9. Determine device battery capacity consumption for 10 minutes as described for the idle case.
10. Repeat the previous three steps for all devices.
11. Change the display brightness to 25, 50, and 100% (one at a time) and then repeat steps 8 and 9.
12. If the display supports multiple refresh rates, change the refresh rate and then repeat steps 8 and 9.
13. Enable all devices.
14. Save the calibration values to an XML file.

(*Note: the sleep parameter chosen were based after conducting extensive testing where we found the minimum sleep value by which all subsequent collection resulted in the same calibration values. We rounded up the value in order to account for possible error rates among different platforms.*)

Finally, we can determine the power consumption of each device. Since our battery capacity is in milliwatts per hour, we can convert it to joules as shown in Equation 2. Then, we can determine the platform's average power consumption by applying Equation 3, where *d* is the duration in seconds. We conclude by calculating the power savings (gain) P_{Saving} of the platform device by applying Equation 4, 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.

$$\Delta C_X = C_{Xt_0} - C_{Xt_1} \tag{1}$$

$$E_{(j)} = \Delta C_{(mWh)} \times \underbrace{\frac{1}{1000}}_{\text{Convert to Watt}} \times \underbrace{3600}_{\text{Convert to Seconds}} \tag{2}$$

$$P = \frac{E_{(j)}}{d_{(s)}} \tag{3}$$

$$P_{Saving} = P_{Device} - P_{Idle} \tag{4}$$

$$\Delta E_{MSR} = \{E_{t_0} - E_{t_1}\} * U \tag{5}$$

$$E_X = E_{All} * U_X \tag{6}$$

$$Rate_{Avg} = \frac{\Delta C}{\Delta t} \tag{7}$$

$$\text{Life}_{Saving} = \frac{C}{Rate_{Avg} - P_{App}} - \frac{C}{Rate_{Avg}} \tag{8}$$

Energy Profiling Module

The energy profiling module consists of energy profiling the running applications, and the platform devices.

Energy Profiling of Applications: In order to determine the energy consumption of applications, we relied on the Machine Specific Registers (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. We chose Intel processor's because it currently dominates the market share for Windows platforms. Intel processors support four MSRs for Running Average Power Limit (RAPL) [3]. *MSR_RAPL_POWER_UNIT* contains power units, energy status units, and time units, and *MSR_PKG_ENERGY_STATUS* and *MSR_PP1_ENERGY_STATUS*, report package and graphics actual energy. 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, we initialize the driver and read the power unit MSR. At 1-second intervals, we collected the battery capacity, energy MSRs, and running processes. In order to get the list of processes, we got a snapshot of the processes handler. 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, we determined the percentage of active time of the processor and each process. In order to calculate the energy used by package and graphics, we calculated the ΔE_{MSR} based on Equation 5 for each energy MSR and where *U* is the energy unit retrieved from *MSR_RAPL_POWER_UNIT*.

Package MSR energy contains the energy consumed by the cores, graphics, and system agents. As a result, we subtract the energy consumption of graphics from package and assign it to total CPU energy E_{All}. Finally, we allocate the energy consumption of application *X* as E_X as shown in Equation 6, 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 7. We repeat 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 without posing extensive overhead of continuous polling of data. Upon completion, we got the minimum average energy consumption of each application. We also calculated the average discharge rate. Then, to determine the battery life savings upon termination of the application Life_{Saving}, we used Equation 8, 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.

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 9, we can calculate the life savings (or lost) Life_{Saving}, where P_{Saving} is calculated based on Equation 4 and where $Rate_{A}vg$ is the same number as the one calculated during energy profiling of applications.

$$\text{Life}_{Saving} = \frac{C}{Rate_{Avg} - P_{Saving}} - \frac{C}{Rate_{Avg}} \tag{9}$$

User-Interactive Module and Power-Management Module

Upon completion of the energy-profiling module, the user-interface gets updated with all the devices that can be controlled by users and their estimated battery life savings, in addition to the top 5 battery-consuming applications with their estimated battery life saving. Using a checkbox, users can select the devices they want to control and the applications they want to terminate. Then, the power-management module is triggered to change the device's state. 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 user's selection. 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 10, where $\text{Life}_{Current}$ 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.

$$\text{Life}_{Req} = \text{Life}_{Current} + \text{Life}_{Min} \tag{10}$$

Monitoring Module

In order to monitor battery life, a new thread is created. At a 2-minute interval, it collects the expected battery life. It subtracts interval time *i* from Life_{Req} as shown in Equation 11 and then compares the value to the expected battery life collected. If after five iterations, Life_{Req} is less than expected battery life, the *Energy Profiling Module* is triggered again where the previously described steps are repeated. If there are not five consecutive errors, the monitoring module continues until the Life_{Req} reaches 2 minutes.

$$\text{Life}_{Req} = \text{Life}_{Req} - i \tag{11}$$

EXPERIMENTAL ANALYSIS

Validating BatteryExtender consists of validating the tool in terms of reconfiguration of the hardware device components, and in terms of energy profiling of applications in our lab. For the first part, we ran scenarios using the two platforms as described in Tables 2 and 3. Prior to running our experiments, we set the power-management policy to default settings and terminated all (foreground and background) applications with the exception of BatteryExtender. 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" and changed the configuration based on scenario (For example, when Wi-Fi is not needed, we disabled it.) We chose "10 minutes" as opposed to a different duration because our goal was to determine: (1) whether we can save battery by examining the amount of battery capacity saved, and (2) the accuracy of our tool by comparing expected and actual capacity savings.

During our experiments, we collected the battery metrics using our "Battery Usage". We started the collection at the start of the test scenario, and stopped upon completion. We calculated **actual total capacity used** (energy used) by test case *X* denoted as ΔC_X as shown in Equation 1 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_{savX} as shown in Equation 12 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 $\text{E}_{\text{sav}_\text{X}}$ as shown in Equation 13 where *n* is the total number of disabled or modified devices and $\text{E}_{\text{sav}_\text{D}}$ is expected capacity savings for device *D* for duration d_base in minutes. We calculated the **total expected capacity savings** for scenario *X* denoted as $\text{ET}_{\text{sav}_\text{X}}$ as shown in Equation 14 where d_X is the total test duration in minutes.

$$AT_{savX} = \Delta C_{\text{DF}} - \Delta C_{\text{BE}} \tag{12}$$

$$E_{\text{sav}_\text{X}} = \sum_{D=1}^{n} E_{\text{sav}_\text{D}}, d_{\text{base}} = 10 \tag{13}$$

$$ET_{\text{sav}_\text{X}} = \frac{E_{\text{sav}_X} \times d_X}{d_{\text{base}}} \tag{14}$$

In order to validate BatteryExtender in terms of energy profiling of applications, we profiled the energy used by the applications using BatteryExtender and then terminated the applications. Next, we observed the impact on the platform's overall energy consumption.

Calibration Results

Table 4 represents the data collected for Dell and Surface. All devices in the table are self-explanatory with the exception of "USB Root Device (xHCI)". By disabling this device, we disable USB input, HID Sensor Collection, the camera, and the Bluetooth adapter. We noticed the same components, on different platforms, can consume different battery capacity.

Download Scenario

During the download scenario, we used Amazon Unbox Video Player [1] to download a movie of size 1.91 GB. For the Default (DF) case, we started BatteryExtender. Then, using the "battery usage" UI, we collected battery capacity remaining at a 30-second interval. Next, we started Amazon Player, selected the movie, and pressed on download. Upon completion, 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. We disabled components (change in the case of display) as shown in Table 5. Then, we collected the metrics as described for the DF case.

The results comparing DF versus BE for Dell are displayed in Figure 4. 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

Device Components	Dell Convertible	Surface 2 Pro
Wi-Fi	120	52
Bluetooth	29	29
NFC	5	N/A
HID Sensor Collection	10	8
VS Location Simulator	10	8
Touchscreen Sensor	10	7
Pen Sensor	N/A	7
Audio	10	12
Front Camera	20	5
Rear Camera	N/A	5
Printer Queue	10	10
USB Root Device xHCI	80	N/A
Refresh @ 59 Hz	20	N/A
Brightness @ 25	190	136
Brightness @ 50	80	70
Brightness @ 100	-(110)	-(207)

Table 4. Calibration results: Capacity saved in mWh during 10 minutes duration using idle at 75% brightness as the base.

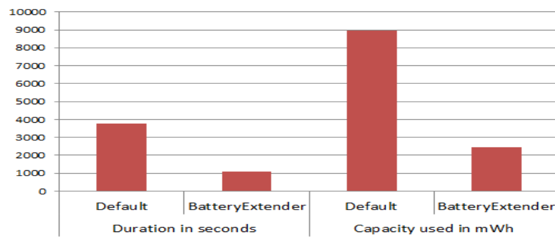

Figure 4. Battery capacity over time during download scenario for Dell Convertible.

duration during BE scenario took 1080 seconds (18 minutes), 71.2 % 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 and memory utilization jumps from an average of 10% and 32% to 18% and 55%, respectively. Moreover, 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 for Surface are displayed in Figure 5. 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.

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 disabled components (changed in the case of display), as shown in Table 6. The results comparing DF versus BE for Dell are displayed

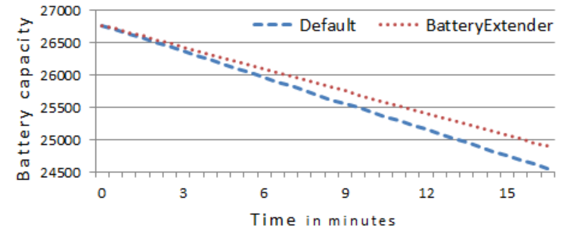

Figure 5. Battery capacity over time during download scenario for Surface 2 Pro.

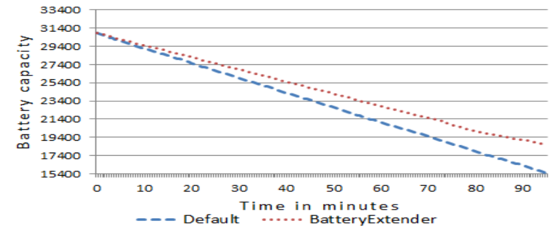

Figure 6. Battery capacity over time during video playback scenario for Dell Convertible.

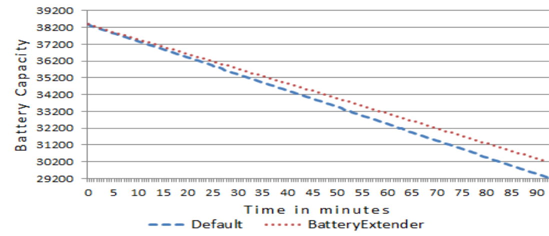

Figure 7. Battery capacity over time during video playback scenario for Surface 2 Pro.

in Figure 6. Energy consumed during DF ΔC_{DF} is 15,450, whereas the ΔC_{BE} is 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 for Surface are displayed in Figure 7. 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.

Video Streaming Scenario

The video streaming scenario consists of streaming a 10 minutes video using YouTube. We ran both cases DF and BE and collected the battery metrics as described in the previous scenario and disabled/changed devices as shown in Table 7. The results comparing DF versus BE for Dell are displayed in Figure 8. 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. CPU and memory activities spiked due to the security service. As a result, the energy savings far exceeded our expectations.

Disabled Devices	Dell Convertible	Surface 2 Pro
NFC	5	N/A
Bluetooth	N/A	29
VS Location Simulator	10	8
HID Sensor Collection	N/A	8
Pen Sensor	N/A	7
Rear and Front Cameras	N/A	10
Touchscreen Sensor	10	7
Audio	10	12
Printer Queue	10	10
USB Root Device (xHCI)	80	N/A
Refresh @ 59 Hz	20	N/A
Brightness @ 25	190	136
Expected Savings in 10 minutes	**335**	**227**

Table 5. Disabled devices and display settings associated with expected capacity savings in mWh during download scenario.

Disabled Devices	Dell Convertible	Surface 2 Pro
WiFi	120	52
Bluetooth	N/A	29
NFC	5	N/A
HID Sensor Collection	N/A	8
VS Location Simulator	N/A	8
Pen Sensor	N/A	7
Touchscreen Sensor	10	7
Printer Queue	10	10
USB Root Device (xHCI)	80	N/A
Refresh @ 59 Hz	20	N/A
Brightness @ 50	80	N/A
Read and Front Camera Rear	N/A	10
Expected Savings in 10 minutes	**325**	**131**

Table 6. Disabled devices and display settings associated with expected capacity savings in mWh during video playback scenario.

Disabled Devices	Dell Convertible	Surface 2 Pro
NFC	5	N/A
Bluetooth	N/A	29
HID Sensor Collection	N/A	8
VS Location Sensor	N/A	8
Pen Sensor	N/A	7
Touchscreen Sensor	10	7
Printer Queue	10	10
USB Root Device (xHCI)	80	N/A
Refresh @ 59 Hz	20	N/A
Brightness @ 50	80	70
Rear and Front Camera	N/A	10
Expected Savings in 10 minutes	**205**	**149**

Table 7. Disabled devices and display settings associated with expected capacity savings in mWh during video streaming scenario.

The results comparing DF versus BE for Surface are displayed in Figure 9. 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.

Validating Energy Profiling of Applications

In order to validate BatteryExtender in terms of energy profiling of applications, we used Surface. We chose to extend battery life for "10 minutes." 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. 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. 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.

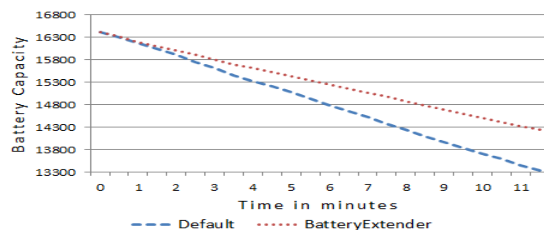

Figure 8. Battery capacity over time during video streaming scenario for Dell Convertible.

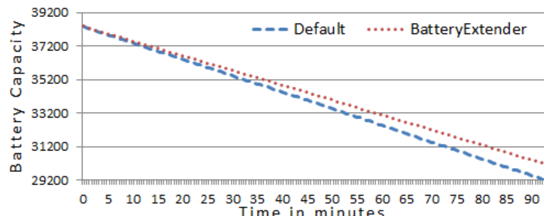

Figure 9. Battery capacity over time during video streaming scenario for Surface 2 Pro.

CONCLUSION AND FUTURE WORK

We presented BatteryExtender, a tool to extend battery life on demand. It enables the reconfiguration of mobile devices in order to utilize only the resources required for specific tasks. It also provides an estimate of the impact of applications on the overall battery life. Our lab results showed a reduction of energy consumption between 10.03% and 20.21%, depending on the workload. The accuracy rate ranged between 92.37% and 99.72%. In addition, in some rare cases, we were able to reduce energy consumption by 72.83% due to the platform's inefficient security service. In the future, we are planning to include field testing with multiple users with various technological skills in order to improve the tool's usability. We are also planning on improving our resource power-consumption estimation by continuously profiling the platform when battery life is not limited. Finally, we are planning on adding the dependency between hardware modules and between applications.

ACKNOWLEDGMENT

This work is in part supported by NSF grant CNS-1205338 and the Introduction of Innovative R&D team program of Guangdong Province (NO. 201001D0104726115). This material is based upon work supporting while serving at the National Science Foundation.

REFERENCES

1. Amazon unbox vido player. `http://www.amazon.com/gp/video/ontv/player`.
2. Device power management. `http://msdn.microsoft.com/en-us/library/windows/hardware/dn495664(v=vs.85).aspx`.
3. 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`.
4. Carroll, A., and Heiser, G. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (2010), 21–21.
5. Consolvo, S., McDonald, D. W., Toscos, T., Chen, M. Y., Froehlich, J., Harrison, B., Klasnja, P., LaMarca, A., LeGrand, L., Libby, R., 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*, ACM (2008), 1797–1806.
6. Dong, M., Choi, Y.-S. K., and Zhong, L. Power modeling of graphical user interfaces on oled displays. In *Proceedings of the 46th Annual Design Automation Conference*, ACM (2009), 652–657.
7. Falaki, H., Mahajan, R., and Estrin, D. Systemsens: a tool for monitoring usage in smartphone research deployments. In *Proceedings of the sixth international workshop on MobiArch*, ACM (2011), 25–30.
8. Hofemeier, G. Ultrabook and tablet windows* 8 sensors development guide. `http://software.intel.com/en-us/articles/ultrabook-and-tablet-windows-8-sensors-development-guide`, 2013.
9. Kang, J., Park, M., Lee, C., and OH, S. User interface method and apparatus therefor, Jan. 9 2014. US Patent App. 13/928,919.
10. Kansal, A., Saponas, S., Brush, A., McKinley, K. S., Mytkowicz, T., and Ziola, R. 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*, ACM (2013), 661–676.
11. Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. T. A survey of mobile phone sensing. *Communications Magazine, IEEE 48*, 9 (2010), 140–150.
12. Maker, F., Amirtharajah, R., and Akella, V. Update rate tradeoffs for improving online power modeling in smartphones. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, IEEE (2013), 114–119.
13. Metri, G., Agrawal, A., Peri, R., and Shi, W. What is eating up battery life on my smartphone: A case study. In *Energy Aware Computing, 2012 International Conference on*, IEEE (2012), 1–6.
14. Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, ACM (2012), 317–328.
15. Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., and Boda, P. 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*, ACM (2009), 55–68.
16. Oliner, A., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. Carat: Collaborative energy diagnosis for mobile devices.
17. Pathak, A., Hu, Y. C., and Zhang, M. 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*, ACM (2012), 29–42.
18. Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y.-M. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, ACM (2011), 153–168.
19. Rivoire, S., Ranganathan, P., and Kozyrakis, C. A comparison of high-level full-system power models. *HotPower 8* (2008), 3–3.