Analyzing Mobile Application Software Power Consumption via Model-Driven Engineering

Chris Thompson, Jules White, and Douglas C. Schmidt
Vanderbilt University, Nashville, TN USA
Email: {chris, jules, schmidt}@dre.vanderbilt.edu

Insitute for Software Integrated Systems

Smartphones are mobile devices that travel with their owners and provide increasingly powerful services. The software implementing these services must conserve battery power since smartphones may operate for days without being recharged. It is hard, however, to design smartphone software that minimizes power consumption. For example, multiple layers of abstractions and middleware sit between an application and the hardware, which make it hard to predict the power consumption of a potential application design accurately. Application developers must therefore wait until after implementation (when changes are more expensive) to determine the power consumption characteristics of a design.

This paper provides three contributions to the study of applying model-driven engineering to analyze power consumption early in the lifecycle of smartphone applications. First, it presents a model-driven methodology for accurately emulating the power consumption of smartphone application architectures. Second, it describes the System Power Optimization Tool (SPOT), which is a model-driven tool that automates power consumption emulation code generation and simplifies analysis. Third, it empirically demonstrates how SPOT can estimate power consumption to within $\sim$3-4% of actual power consumption for representative smartphone applications.

1 INTRODUCTION

Emerging trends and challenges. Smart devices, such as smartphones and tablet computers have been steadily increasing in capabilities and performance. Modern devices commonly contain multi-core CPUs running at clock speeds well over 1GHz. For example, the Galaxy Nexus from Google contains a 1.2 GHz, dual-core processor with 1GB of RAM and other Android smartphones now have quad-core processors.

The increased performance of smart devices comes at a cost, however. Due to their mobility, smartphones and tablets run exclusively on battery power, so every increase in performance must be balanced with an increase in battery capacity to maximize the time between charges. The design of applications running on the device can also impact battery life, *e.g.*, frequently sampling the GPS or making network requests can consume a significant amount of power.

To balance functionality with power consumption effectively, developers must understand various trade-offs. A key trade-off is between performance and battery life, as well as the implications of the application and infrastructure software architecture on power consumption. Functional requirements, such as minimum application response time, can conflict with power consumption optimization needs.

For example, a traffic accident detection application [29] must be able to detect sudden accelerations indicative of a car accident. To detect acceleration events that indicate accidents, the application must sample device sensors and perform numerous calculations at a high rate. Conflicts occur between the *functional requirements* (*e.g.*, the minimum sensor sampling rate needed to accurately detect accidents) and *non-functional requirements* (*e.g.*, sustaining operations on the mobile device without frequent battery recharging).

Due to complex middleware, operating system (OS), and networking layers, it is hard to predict the effects of application software architecture decisions on power consumption *a priori*, *i.e.*, without actually *implementing* a design, which makes it hard to analyze the power consumption of design until late in the development cycle, when changes are more expensive [10]. For example, a developer may elect to use HTTPS instead of HTTP to satisfy a security requirement by making communication between the application and server more confidential. It is currently hard, however, to predict how much additional power is consumed by the added encryption and decryption of data without actually implementing the system.

It is also hard to quantify the trade-off between power consumption and security, as well as many other design decisions. Moreover, certain design decisions, such as data transmission policies (*e.g.*, should an application transmit immediately or wait for a specific medium like a Wi-Fi or 3G cellular connection) are hard to analyze early in the design cycle, due to their variability. For example, if an application only sends data intermittently, it may be beneficial to transmit small amounts of data over cellular connections due to the decreased power consumption of 3G cellular connection compared to Wi-Fi [1]. If a large amount of data must be sent, however, the total time required to transmit it over 3G may negate the benefit of using the less power consumptive connection. The cellular connection will take longer to transmit the data, therefore, which may in turn consume more total power than the Wi-Fi radio that can transmit the data faster.

Solution approach → Power consumption emulation of mobile software architectures with model-driven testing and auto-generated code. By using *Model-driven Engineering* (MDE) [22], we allow developers to specify a *domain-specific modeling language*(DSML) [17] to capture key software architecture elements related to power consumption. Developers can then use automated code generators to produce emulation code from this model. Developers can also execute the generated emulation code on target hardware, collect power consumption information, and analyze the application's power consumption prior to actual deployment.

This emulation code allows developers to analyze proposed software architectures prior to investing significant time and effort in a complete implementation of the design. The auto-generation of emulation code also enables developers to compare a range of different designs quantitatively during initial phases of a development process [9], which allows developers to select designs that satisfy both functional and non-functional requirements while minimizing overall power consumption. This analysis can also occur early in the software lifecycle (*e.g.*, at design time), thereby avoiding costly changes to optimize power consumption later downstream.

This paper describes the *System Power Optimization Tool* (SPOT), which uses MDE techniques to analyze the power consumption of mobile software architectures. SPOT allows developers to create high-level models of candidate software architectures using the *System Power Optimization Modeling Language* (SPOML) that capture key software components related to power consumption. SPOT generates emulation code from the model that can be executed on target devices.

As the SPOT-generated emulation code runs it is instrumented to collect power consumption data. This data can later be downloaded and analyzed offline. Developers can apply this emulation and analysis cycle to understand power consumption implications of their designs *without* expensive and time consuming manual programming using third-generation languages, such as C#, C/C++, and Java.

SPOT's generated emulation code mimics key power-consuming aspects of a proposed software architecture. Key power consumptive components of mobile application software architectures include GPS, acceleration, orientation, sensor data consumers, and network bandwidth consumers [21]. Focusing SPOT on these components allows developers to model the most significant power expenditures of their applications. Moreover, as applications are constructed, the generated emulation code can be replaced incrementally with actual application components, allowing developers to refine the accuracy of the analysis continuously throughout the software lifecycle.

This chapter extends our prior work on model-driven application power consumption analysis [25] by providing the following new contributions: (1) surveying related work on optimizing power consumption, domain-specific power optimizations, software-based power consumption estimation, and mathematical power estimation models (2) quantitatively analyzing the accuracy of a SPOT model based on only key power-consuming components and showing how modeling only power consuming components allows developers to avoid the accidental complexities of non-power consuming architectural aspects, (3) discussing the implications of the simplified model on application software architecture and design, (4) empirically evaluting the effects of design decisions, such as sensor sample rate, on overall application power consumption, and (5) describing how data produced by SPOT can be used to refine and optimize application power consumption.

Paper organization. The remainder of this paper is organized as follows: Section 2 compares SPOT with related work on optimizing power consumption; Section 3 outlines a motivating example we use to showcase and evaluate SPOT's functionality throughout the paper; Section 4 summarizes the challenges associated with predicting power consumption of mobile application software architectures; Section 5 describes the structure and functionality of SPOT and SPOML; Section 6 empirically evaluates SPOT's power prediction capabilities and shows how its modeling primitives and emulation infrastructure can accurately predict power consumption for representative mobile applications on the Android smartphone platform; and Section 7 presents concluding remarks.

2 Survey of Related Work

This section provides a survey of prior work that compares SPOT with related research in the following categories:

- **System execution modeling tools**, which model and evaluate performance related information early in an application's development process,
- **Domain-specific power optimizations**, which are approaches for reducing the power consumption of specific application functions, such as geo-localization,
- **Power instrumentation and mathematical estimation of power consumption**, which attempt to directly measure or predict power consumption of an application,
- **Hardware-based power consumption optimization**, which involves specialized firmware or low power physical hardware,
- **Network protocol optimizations** are made in the network stack to reduce the overall amount of data transmitted as well as contention on the physical link,
- **Post-implementation power consumption analysis** requires developers to completely implement an application before analyzing its power consumption.
- **Model-driven engineering** which uses domain-specific modeling languages and code generation to reduce development time and cost.

As discussed below, these methods of power consumption optimization are effective at reducing overall power consumption and are complimentary to SPOT. SPOT does not optimize the power consumption of an application, but rather allows developers to analyze the power consumption of key power consuming elements of an architecture at design-time.

System execution modeling tools. The *Component Utilization Test Suite* (CUTS) [9] is a system execution modeling tool [23] that allows developers to model key architectural elements of distributed systems that determine performance. CUTS allows developers to model a distributed software architecture and then generate emulation code to predict performance. Although CUTS and SPOT share common domain-specific modeling patterns, CUTS's modeling abstractions focus on performance-related aspects of distributed systems, whereas SPOT's modeling abstractions focus on important power consumers in mobile applications, such as GPS usage. Moreover, it is not possible to capture power consumption information from CUTS emulation code or to generate mobile emulation applications for Android or iPhone.

Domain-specific power optimizations. Prior work has examined domain-specific power optimizations that are geared towards a specific type of software feature. For example, Kjasrgaard et al. [12] have evaluated specialized approaches for optimizing the power consumption of location-based services using both client and server modifications. They propose lowering the accuracy of position information based on application-specific information, such as the current zoom level of a map. Other optimizations they propose include caching position and then continuing to report that position as long as the device is expected to be within a specified threshold of that cached position.

Many optimizations proposed by Kjasrgaard et al. are obtained through power profiling of applications. SPOT is complementary to this work and provides tools for profiling software applications early in their lifecycle. SPOT's goal is to aid developers in discovering the types of power optimization strategies described by Kjasrgaard et al.

Another area of mobile power consumption that studied by Thiagarajan et al. [24] is the energy expended downloading and rendering web pages using a mobile browser. They have conducted extensive instrumentation of mobile devices and webkit-based browsers to profile exactly how much energy is consumed during each phase of accessing a mobile site. For example, they measure the variabilities of energy consumption across popular sites, such as Apple and Amazon, as well as exploring how images, javascript, and overall rendering consume power on the device. Their research highlights the importance of known optimizations, such as minimization of javascript files and scoping of CSS to a particular page, that are not always employed as they should be by developers.

The work by Thiagarajan et al. is synergistic with SPOT and looks at power consumption within an individual existing application on a device. SPOT also facilitates the collection of power consumption data from applications, but earlier in the design cycle when the implementation is may not be complete. Both prior approaches aid developers in improving design decisions to reduce mobile power consumption but are focused on different domains (*e.g.,* mobile website design vs. early analysis of mobile applications).

Context-aware power optimizations have been proposed for sensor networks to ensure that sensors are only sampled in contexts where it makes sense. Herrmann et al. [8] propose using optimizations, such as detecting indoor and outdoor transitions to determine when an outdoor sensor should be used. They demonstrate through modeling that these types of context-based optimizations can have significant power savings. SPOT helps evaluate these types of context-aware power optimizations by allowing developers to model architectures that exhibit different optimizations and experiment with them early in the design cycle.

Due to the high power consumption of many mobile sensors, another important area of research involves optimizing sensor sampling rates and times to minimize power consumption. For example, Wang et al. [28] use Markov models to determine the best sampling approaches for sensor networks based on smartphones. These approaches are orthogonal to SPOT and focus on a different aspect of power optimizations. SPOT focuses on generating applications that behave like the planned architectures of specific mobile applications to facilitate early experimentation and reasoning about power consumption.

Power instrumentation. A challenge on mobile devices is accurately collecting detailed and real-time power consumption information without external hardware. Yoon et al. [32] have proposed a more accurate and real-time framework for monitoring power consumption on Android devices. Their approach, called AppScope, intercepts key system calls to the underlying operating system, such as calls to the Android Binder, and then use information about the types and frequency of calls being made to infer power usage.

AppScope could be used to further improve the power consumption predictions made by SPOT. In particular, SPOT focuses on synthesizing mobile application code to run experiments that mimic how a particular mobile application architecture will behave. These synthetic applications can then be used to profile the app and collect power consumption information. Yoon et al.'s framework could be used in place of the Android APIs leveraged by SPOT to improve the fidelity of the power consumption data obtained from running the synthetic applications.

Mathematical estimation of power consumption. Mathematical models of power consumption have been developed to help estimate the cost of using smartphone sensors, such as GPS, and processing capabilities. Kim et al. [11] have developed a mathematical model for estimating the power consumption of smartphones with multicore CPUs. These types of mathematical analyses share a similar aim to SPOT and attempt to provide early prediction of power consumption before the implementation of an application may be completely ready. SPOT, however, assumes that real-world power consumption may vary substantially between devices and may depend on application-specific design details.

Hardware-based power optimizations. Conventional techniques for reducing mobile device power consumption have focused on hardware- or firmware-level modifications to achieve the target consumption [21]. These techniques are highly effective, but are limited to environments in which the hardware can be modified by the end user or developer. Moreover, these modifications tend to result in a tightly coupled architecture which makes them hard to use outside of the context in which they were developed. In other words, a solution might reduce power consumption for one application or in one environment, but may not have the same effect if any of those assumptions change. Moreover, such modifications are typically beyond the scope of software engineering projects and require substantial understanding of low-level systems.

In some instances, hardware-level modifications can actually hurt power consumption by increasing overhead. These techniques are also useful for reducing overall power consumption but do not help in power consumption analysis that is necessary when developing power-conscious applications. SPOT is complimentary to these approaches in that developers can use SPOT to identify the most effective method to minimize power consumption without requiring extensive hardware knowledge or restricting the optimizations a single hardware platform.

Network protocol and interface optimization. Due to the limited battery power available to mobile and embedded systems, much prior work has focused on optimizing system power consumption. In particular, network interfaces consume a large portion of overall device power [13], so reducing the power consumption of networking components has been a research priority. Ultimately, the amount of power consumed is directly proportional to the amount of data transmitted [6]; in some instances require 100 times the power consumed by one CPU instruction to transmit one byte of data [18]. The power consumption of the network interface can thus be reduced by reducing the amount of data transmitted. Moreover, utilizing MAC protocols that reduce contention can significantly reduce power consumption [4]. While MAC-layer modification is effective, it is typically beyond the scope of software-engineering projects, which is common with mobile application development.

Using a custom MAC protocol or specifically selecting a pre-existing one based on the needs of the network may be possible when designing a custom sensor network, it is not possible when developing an application for a pre-existing platform, such as Android. Similarly, other work [19] achieved significant power reductions by using middleware to optimize the data transmitted. These approaches require both the device and any access points to which it connects to implement the same MAC protocol, which is infeasible for mobile application developers. Other work accomplished similar goals through transport layer modifications [15], but the same limitations apply as with modifications to the MAC layer.

SPOT seeks to accomplish similar goals by modifying the data transmitted by the application layer, rather than attempting to modify the underlying network stack. SPOT helps developers analyze the data they transmit to maximize throughput of relevant data (*e.g.* actual data versus markup or framing overhead) thereby reducing power consumption. In addition, SPOT can function in a complimentary role allowing developers to analyze the power consumption of network protocol optimizations to identify the most effective configuration.

Post-implementation power consumption analysis. Previous work [5] not only identified software as a key player in mobile device power consumption (*e.g.*, Symbian devices), but also sought to allow developers to analyze the power consumption of applications during execution. Moreover, other work [16] utilized a similar approach to analyze the power consumption of sensor nodes in a wireless sensor network.

While post-implementation power consumption analysis can provide highly accurate results, it suffers from the pitfalls of post-implementation testing, including increased cost of refactoring if problems are discovered. To prevent costly post-implementation surprises, SPOT allows developers to analyze designs before any code is written. It also allows them to perform continuous integration testing through the use of custom code components to further refine the accuracy of the model as development progresses.

Model-driven engineering. Modeling and code generation have been combined in a number of model-driven engineering [22] approaches to reduce the number of development errors and required manual code creation. Prior work has investigated environments, such as GME [17] and GEMS [31], for rapidly creating domain-specific modeling language tools and code generation infrastructure. Where as GME and GEMS are generic modeling platforms for creating domain-specific languages and tooling, SPOT provides a specific domain-specific language targeted to the domain of early analysis of mobile architecture power consumption. SPOT builds upon the prior work and provides a specific instantiation of the concepts and new and novel techniques targeted at mobile power consumption.

3 Motivating Example: the WreckWatch Case Study

This section describes *WreckWatch* [29], which is an open-source[1] mobile application we built on the Android smartphone platform to detect automobile accidents. We use WreckWatch as a case study throughout this paper to demonstrate key complexities of predicting the power consumption of mobile software architectures. As shown in Figure 1, WreckWatch operates by (1) monitoring smartphone sensors (such as GPS receivers and accelerometers) for sudden acceleration/deceleration events that are indicative of an accident. Data about the event are then (2) uploaded to a remote server over HTTP where first-responders and/or other motorists can access the information via a web browser or Android client. WreckWatch allows bystanders (3) to upload images of the accident to the same web server, thereby increasing the information first-responders possess before arriving at the scene.

Information, such as images of the accident and force of impact, provides first-responders with immediate knowledge of an accident. It also seeks to provide them

[1] WreckWatch is available from `vuphone.googlecode.com`.

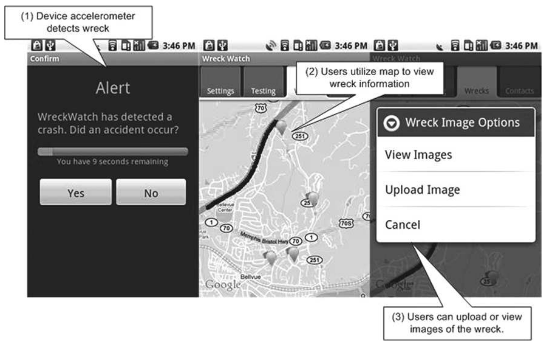

(1) Device accelerometer detects wreck
Confirm
Alert
WreckWatch has detected a crash. Did an accident occur?
You have 9 seconds remaining
Yes
No
Wreck Watch
Settings
Testing
(2) Users utilize map to view wreck information
Wrecks
Contacts
Wreck Image Options
View Images
Upload Image
Cancel
(3) Users can upload or view images of the wreck.

Fig. 1: WreckWatch Operation

with richer situational awareness to better prepare for conditions at the accident scene. Moreover, accident location information helps alleviate potential congestion due to an accident by providing motorists a means to immediately identify potential areas of congestion and avoid the area altogether (*e.g.* this accident location information can be combined with a navigation package, such as the Google Maps API, to alter routing information dynamically so motorists can avoid the accident location). When an accident occurs, both first-responders and emergency contacts specified by accident victims can be notified via integrated VoIP infrastructure, as well as by SMS messages and/or email.

To detect traffic accidents accurately, WreckWatch runs continuously as a background service and continuously consumes a great deal of accelerometer and GPS data. The application must therefore be conservative in its power consumption. If not designed properly, WreckWatch can significantly decrease smartphone battery life.

4 Challenges of Designing Power Conscious Mobile Applications

This section describes the challenges associated with developing power-aware mobile software, such as the WreckWatch application described in Section 3. High-level mobile application development SDKs, such as Google Android or Apple iPhone, simplify mobile application development, but do not simplify power consumption prediction during application design. In fact, the abstractions present in these SDKs make it hard for developers to understand the power implications of software architecture decisions until their designs have been implemented [20], as described in Section 4.1. Interaction with sensors, such as accelerometers or GPS receivers and network interaction, can also result in a significant amount of power consumption, as described in Sections 4.2 and 4.3.

4.1 Challenge 1: Accurately Predicting Power Consumption of Framework API Calls

Each line of code executed results in a specific amount of power consumed by the hardware. In the simplest case, this power consumption results from a small series of CPU operations, such as reading from memory or adding numbers. In some cases, however, a single line of code can result in a chain of hardware interactions, such as activation of the GPS receiver and increasing the rate at which the screen is redrawn. Moreover, although the higher levels of abstraction provided by modern smartphone SDKs make it easier for developers to implement mobile application software, they also complicate predictions of the effects on the hardware.

For example, WreckWatch heavily utilizes the Google Maps API and the "MyLocation" map overlay, which provides end users with a marker indicating their current GPS location. The use of the "MyLocation" is typically accomplished with fewer than 10 lines of code, but results in substantial power consumption. This is because the overlay is redrawn at a high rate to achieve a "flashing" effect, and because the overlay enables and heavily utilizes the GPS receiver on the device, which further increases power expenditure. It is hard to predict how using arbitrary API calls, such as this overlay, will affect application power consumption without implementing a particular design and testing it on a particular target device.

This abstraction in code makes power-consumption analysis on arbitrary segments of code hard. Predicting power usage from high-level design abstractions, such as a UML diagram, is even harder. Section 5.3 describes the MDE and emulation approach we use to address this challenge.

4.2 Challenge 2: Accurately Predicting Power Consumption of Sensor Usage Architectures

In applications utilizing sensor data, the most accurate sensor data is obtained by sampling as often as possible. Sampling at high rates, however, incurs high power consumption [14] by not allowing sensors to enter low power modes and by increasing the amount of data processed by applications. Reducing the sample rate can decrease application power consumption considerably, but also reduces accuracy. The trade-offs between power consumption and accuracy at a given sampling rate are hard to determine without empirical tests on a target device due to the high-degree of layering in modern smartphone APIs and system architectures.

For example, WreckWatch was originally designed to collect GPS data every 500 milliseconds and consume accelerometer data at Android's predefined *NORMAL* rate setting. During the development of WreckWatch, it was clear that reducing WreckWatch's GPS sampling rate would reduce overall power consumption, but it was unclear to what degree. Moreover, it was hard to predict what sample rate would provide sufficient accuracy and still allow the phone to operate for days between charges. Section 5.2 describes how we use automatic code generation to create emulated applications that accurately analyze the power consumption of a candidate sensor utilization architecture without incurring the substantial time and effort to manually implement the architecture.

4.3 Challenge 3: Accurately Assessing the Effects of Different Communication Protocols on Power Consumption Prior to Implementation

Each application and network communication protocol has a specific overhead associated with it and can result in significant power consumption [7]. Certain protocols require more development overhead to implement, but have low runtime overhead (*e.g.* bandwidth consumption, message processing time, etc.). For example, communicating with UDP packets requires implementing flow control, error detection, etc., whereas TCP communication requires setup and teardown time for communication channels, as well as the processing and data associated with these operations, in the form of additional protocol message overhead and processing logic.

More advanced and robust protocols, such as HTTP, are easier to use as communication protocols and are more attractive to developers despite the increased overhead [2]. For example, communicating over HTTP simplifies establishing a communication channel between client and server. There are also many libraries available to extract data from an HTTP message. HTTP supports a great deal of functionality that may not be necessary for a particular application, however, and using this functionality incurs extra overhead that consumes more power.

For a small number of messages the overhead of HTTP will likely have little impact on application power consumption. With large numbers of operations, however, the overhead of HTTP can have a significant impact on power consumption. It is hard to determine early (*e.g.*, at design time) in an applications lifecycle, however, how this overhead will affect power consumption and whether the number of messages transmitted will be substantial enough to impact power consumption significantly. This challenge is exacerbated if certain network operations consume more power than others, *e.g.*, receiving data often consumes more power than transmitting data [27].

For example, to provide the most accurate situational awareness to first responders—and provide the most accurate congestion information to motorists—the WreckWatch application must periodically request wreck information from the central web server. These updates must be done periodically and were originally intended to run over HTTP. Using HTTP results in a significantly less developer effort but results in a considerable amount of communication overhead from the underlying TCP and HTTP protocols, which ultimately transmits substantial amounts of data that have no relevance to the application. It is hard to determine at design time if/how this additional data transmission will significantly impact power consumption. Section 5.2 shows how we used MDE code generation to implement and analyze potential communication protocols rapidly.

5 The System Power Optimization Tool (SPOT)

This section describes the structure and functionality of the *System Power Optimization Tool* (SPOT), which is an MDE tool that allows developers to model potential mobile application software architectures to predict their power consumption, generate code to emulate that architecture, and then systematically analyze its power consumption properties. SPOT addresses the challenges described in Section 4 by allowing developers to understand the implications of their software architecture decisions at design time.

SPOT's development process enables developers to generate visual, high-level models rapidly, as shown in Figure 2. These models can then be used to analyze the power consumption of mobile application software architectures (step 1 of Figure 2). SPOT thus helps overcome key challenges of predicting power consumption by generating

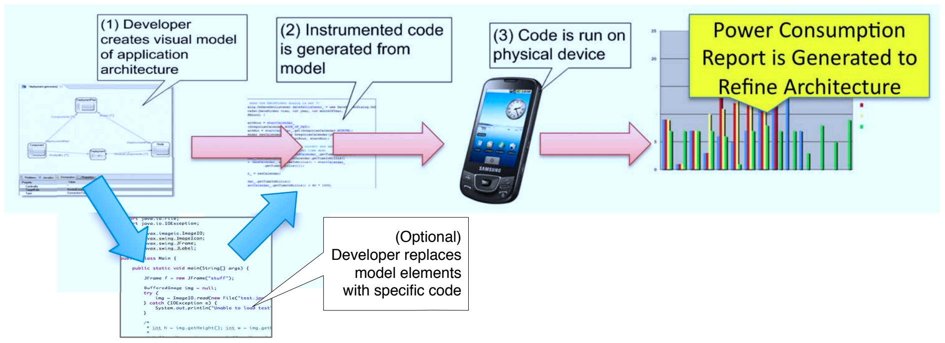

(1) Developer creates visual model of application architecture
(2) Instrumented code is generated from model
(3) Code is run on physical device
Power Consumption Report is Generated to Refine Architecture
(Optional) Developer replaces model elements with specific code

Fig. 2: SPOT Modeling and Analysis Process

device logic that can be used to gather power consumption information on physical hardware during early phases of an application's software lifecycle, which helps minimize expensive redesign/refactoring costs in later phases.

SPOT uses models shown in Figure 2 to generate instrumented emulation code for the given platform or device (step 2 of Figure 2). When executed on actual hardware (step 3 of Figure 2), this generated code collects power consumption and system state information. This power consumption data can then be downloaded and analyzed offline to provide developers with application power utilization at key points throughout the lifecycle and execution path (step 4 of Figure 2).

SPOT also supports the use of custom code modules. These models allow developers to replace automatically generated code with actual application logic while still providing the same analytical capabilities available when using generated code. SPOT therefore not only allows developers to perform analysis early in the development cycle, but also to perform continuous integration testing throughout development.

SPOT is implemented as a plugin for the Eclipse IDE. Its runtime power consumption emulation and capture infrastructure is built using predefined, user-configurable classes that emulate specific power consuming components, such as GPS data consumers. This infrastructure captures power consumption information during executing by using the device's power API. For example, power data collection on the Android platform is performed by interfacing with the OS application power API, *i.e.*, the Android power consumption API as implemented by the "FuelGauge" application.

The remainder of this section describes how SPOT's DSML, emulation code generation, and performance measurement infrastructure help application developers address the challenges presented in Section 4. Section 5.1 describes SPOT's modeling language, SPOML, Section 5.2 describes how SPOT generates emulation code, and Section 5.3 describes how SPOT analyzes and evaluates emulation code during execution.

5.1 Mobile Application Architecture Modeling and Power Consumption Estimation

SPOT describes key power-consuming aspects of a mobile application via a DSML with specific language elements. This DSML allows developers to specify their software architecture visually with respect to power consuming components, as shown in Figure 2.

Prior work [25, 29, 26] showed how the following components are often significant power consumers in mobile applications:

• **CPU consumers** are used to represent CPU-intensive code segments such as calculations on sensor data. Developers can specify the amount of CPU time that should be consumed by specifying the number of loop iterations of a square root calculation that should be run. For example, WreckWatch developers can model the mathematical calculation time to determine the current G-forces on the phone.

• **Memory consumers** generate dynamically allocated memory. These consumers allow developers to analyze not only the power consumed by actual operations, but also their impact (such as the frequency and duration of garbage collector sweeps) on garbage collection. Developers can specify the amount of memory to consume as bytes. For example, WreckWatch developers can model the effects of caching accident images of varying sizes.

• **Accelerometer consumers**, which interact with system accelerometers and consume accelerometer data. These consumers can be configured to utilize the full range of system-supported sample rates. For example, WreckWatch developers can model the sensor interaction needed to accurately detect car accidents.

• **GPS consumers** interact with the device's GPS receiver. These consumers can be configured with custom sample rates as well as a minimum distance between points, *i.e.*, the sensor will only return a data point if the distance between the current point and the last point is greater than a specified value. GPS consumers allow developers to analyze the impact of using a location service configuration on power consumption. For example, WreckWatch developers use this capability to model how polling for a vehicle's location at different rates impacts power consumption.

• **Network consumers** emulate application network interaction by periodically transmitting and receiving data. Network consumers allow users to supply SPOT with sample data that is then transmitted at the interval specified. For example, WreckWatch developers can provide a URI along with server and port information to configure SPOT to make a specific request. These consumers can also be configured to execute at varying times to emulate periodic updates.

• **Screen drawing agents** utilize graphics libraries, such as OpenGL, to emulate a graphics-intensive application, such as a game or streaming video to a first responder's WreckWatch client. Users can configure these consumers by specifying the types and size of objects to draw on the screen, along with any transformations that should be performed on the object. For example, WreckWatch developers can use the drawing agents to show how the use of images and video for situational awareness impacts battery life.

• **Custom code modules** allow developers to specify their own code to run against the profiling package. This capability allows developers to extend SPOT's functionality to meet their needs, as well as incrementally replace the emulation code with actual application logic as it becomes available. Replacing the emulation logic allows developers to perform testing as development progresses and increase the accuracy of the evaluation and analysis. For example, WreckWatch developers can use these consumers to include a service for uploading multimedia content about an accident to a central web server.

The metamodel for SPOT's DSML, called the *System Power Optimization Modeling Language* (SPOML), allows application developers to build software architectural specifications that determine power consumption from key power consuming components. SPOML was created using the metamodeling features of the Generic Eclipse Modeling System (GEMS) [30], which is a tool for rapidly generating visual modeling tools atop the Eclipse Modeling Framework (EMF). GEMS is built atop Ecore, which provides metamodeling and modeling facilities for Eclipse [3].

Figure 3 shows SPOML's metamodel.

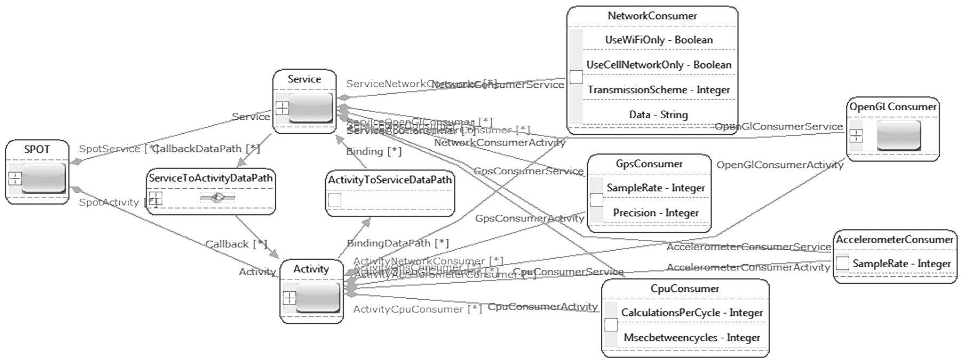


Fig. 3: SPOT's Metamodel

The primary application serves as the root element of the model. Power consumption modules can exist within either *activities* (which are basic building block components for Android applications and represent a "screen" or "view" that provides a single, focused thing a user can do) or *services* (which are background processes that run without user intervention and do not terminate when an application is closed).

Each activity or service can contain one or more power consumer modeling elements described above. Developers can therefore emulate potential decisions that they make when designing a mobile device application, which allows them to emulate a wide range of application software architectures. For example, Figure 4 shows a SPOML model of the WreckWatch application's sensor usage design.

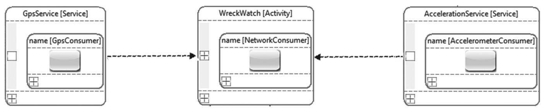


Fig. 4: WreckWatch Model

Acceleration and GPS consumers run in independent services, while the network consumer runs in the application's activity. This model results in an application that monitors GPS and accelerometer values at all times regardless of what the user is doing. It only utilizes the network connection, however, when the user has the WreckWatch application open and running.

5.2 Generating Software Architecture Emulation Code

Predicting the power consumption of an arbitrary design decision is hard, as described in Section 4.1. SPOT addresses this challenge by generating application emulation code automatically to execute on the underlying device hardware. SPOT's automatic generation of emulation code allows application developers to reduce the time required to write enough code to analyze system power consumption accurately. This emulation code is instrumented so the architecture's power expenditures can be examined after a test run.

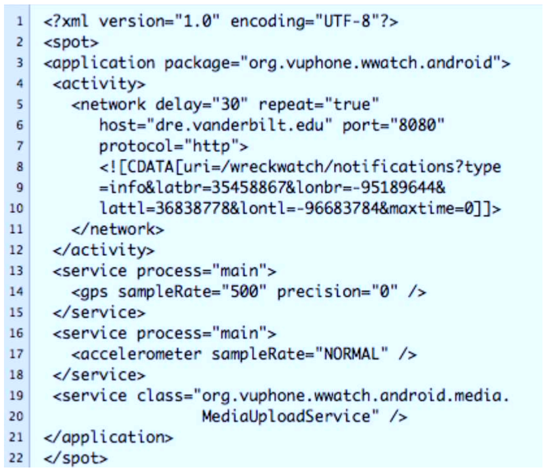
In addition to instrumenting the code, SPOT has the potential to apply the same model for multiple target platforms, such as Android and iPhone, as long as configurable code is available for the power-consuming elements. This emulation and analysis cycle allows developers to observe the power consumption of their design very early in the development cycle, as well as evaluate their software designs across multiple hardware configurations to assess how changes in hardware affect application power consumption. For example, even though the Motorola Droid and Google Nexus One both run the Android platform, each possesses key hardware differences, such as the type and size of the display, that impact power consumption.

The generated emulation code allows developers to address the remaining challenges of selecting an optimal communication protocol and optimizing sensor polling rates, as described in Section 4. Generated emulation code allows developers to evaluate the power consumption of a potential design empirically, rather than simply guessing its power consumption or waiting until an application implementation is complete before running tests. Moreover, developers can quantitatively compare the power consumption effects of choosing different networking protocols and can evaluate the power consumption of different sensor sampling rates.

To accomplish this mobile software architectural emulation, SPOT uses a set of predefined code blocks that can be configured at runtime to perform power consuming operations. SPOT uses an XML configuration file to perform any necessary configuration and create an application that emulates the power consumption of the desired software architecture. To generate this XML configuration file, SPOT interprets the model outlined in Section 5.1. Users of SPOT define the model and tweak configuration parameters, and SPOT can then compile the model and parameters into an intermediate XML format which is utilized to configure prebuilt implementations of each power consuming element described in Section 5.1.

Figure 5 shows a sample of the XML configuration file generated for the WreckWatch model shown in Figure 4. The XML shown in Figure 5 represents a configuration with two background services running an accelerometer consumer and a GPS consumer. The GPS consumer samples every 500 milliseconds, with a minimum radius between sample points set to 0. The accelerometer consumer is set to sample at the *NORMAL* rate, which is a constant defined by Android. There is also a network consumer transmitting sample data every 30 seconds and repeating infinitely. The network consumer is configured to connect to a specific host on a specific port and utilize the HTTP protocol.

The predefined power consumers have configurable options (such as the sample rate of the GPS and data to transmit over the network) provided by the XML configuration file. These power consumption elements are generic and applicable to a wide range of



```xml
<?xml version="1.0" encoding="UTF-8"?>
<spot>
<application package="org.vuphone.wwatch.android">
 <activity>
   <network delay="30" repeat="true"
      host="dre.vanderbilt.edu" port="8080"
      protocol="http">
      <![CDATA[uri=/wreckwatch/notifications?type
      =info&latbr=35458867&lonbr=-95189644&
      lattl=36838778&lontl=-96683784&maxtime=0]]>
   </network>
 </activity>
 <service process="main">
   <gps sampleRate="500" precision="0" />
 </service>
 <service process="main">
   <accelerometer sampleRate="NORMAL" />
 </service>
 <service class="org.vuphone.wwatch.android.media.
                 MediaUploadService" />
</application>
</spot>
```

Fig. 5: Sample WreckWatch Emulation XML

platforms, such as Android, iPhone, or BlackBerry. The predefined power consumers are implemented on the Android platform as follows:

• **CPU consumers** are implemented as a set of nested loops that perform basic mathematical operations, such as square root calculations, on randomly generated data. This module utilizes the CPU while minimizing utilization of other system resources, such as memory that could skew power consumption information. For example, this consumer uses primitive types to avoid allocating dynamic memory. Users can adjust the length of the loops via a configurable parameter. Various end device processors result in same-length loops performing differently on divferent devices, in a manner similar to CPU-intensive algorithmic performance on end-user devices.

• **Memory consumers** are implemented by dynamically allocating custom objects that wrap byte arrays. To analyze the frequency of garbage collection, a Java `WeakReference` object is used to inform the garbage collector that they can be reclaimed, despite having active references within running code. The object's `finalize()` method (which is called immediately before the object is reclaimed by the Android Dalvik virtual machine) is overridden to record the time of garbage collection, thereby allowing developers to analyze the frequency of garbage collection runs. The `WeakReference` object will thus be reclaimed during every garbage collection run.

Due to the limitations of the Android instrumentation API, garbage collection and memory usage must be inferred through analysis of the frequency and duration of garbage collection runs, rather than through direct power consumption measurement. Although this limitation prevents developers from including memory statistics in the data along with CPU, sensor, and network utilization, they can still examine how their design uses memory. Users can also configure the amount of memory and frequency of allocation, as well as supply custom objects (such as WreckWatch's image caches) to use rather than the byte arrays used by default.

• **GPS consumers** are implemented by code that registers to receive location updates from the GPS receiver at specific intervals. To emulate an application's interaction with the GPS receiver properly, SPOT provides developers with all configuration options, such as polling frequency and minimum distance between points, presented by the Android GPS API. This configuration information—along with generic sensor setup and tear-down code—is then inserted into the appropriate location in the emulation application.

• **Accelerometer consumers** are implemented using the configuration specified in the XML file, along with generic setup code to establish a connection to the appropriate hardware. SPOT provides developers with access to device accelerometers and all configuration options the underlying framework presents, such as sample rates.

• **Network consumers** are implemented as emulation code containing a timer that executes a given networking operation, such as an HTTP operation at a user defined interval. The purpose of the network consumer is to emulate applications that use network resources to store information that the phone must periodically retrieve. Developers configure SPOT by providing the data to transmit along with the frequency of transmission. They can also specify whether the data should always be transmitted or whether the application should wait for the availability of a specific transmission medium, such as Wi-Fi or 3G.

• **Screen drawing agents** allow users to specify 3D and 2D graphics to draw on the screen. Developers will specify object contents along with any potential motion or actions.

• **Custom code modules** allow developers to supply their own code blocks to extend the functionality of SPOT and enhance emulation accuracy as the development cycle progresses by substituting the *faux* emulation code with actual application logic. SPOT allows developers to supply class files to load into the emulation application dynamically, as well as method "hooks" to allow the emulation code to interact with the custom code properly.

5.3 Power Consumption Instrumentation

SPOT uses an instrumentation system to capture power consumption statistics as the emulation code is executed, as shown in Figure 6. Components in the instrumentation system are either `Collectors`, `Recorders`, or `Event Handlers`. `Collectors` interface directly with the specific power API on the system and pass collected data to `Recorders`, which persist the data collected by `Collectors` by writing it to a file or other storage medium. `Event Handlers` respond to the events fired by entering or leaving emulation code blocks.

These components are dynamically loaded via Java reflection to ensure extensibility and flexibility. For instance, developers can implement a custom `Collector` to monitor which components are in memory at any given time. Alternatively, developers could define `Recorders` to log power consumption information to another data storage medium, such as a local or network database rather than a flat file.

To analyze an architecture effectively, SPOT records battery state information over time to allow developers to pinpoint specific locations in their application's execution path that demand the most power. To collect power consumption information accurately,

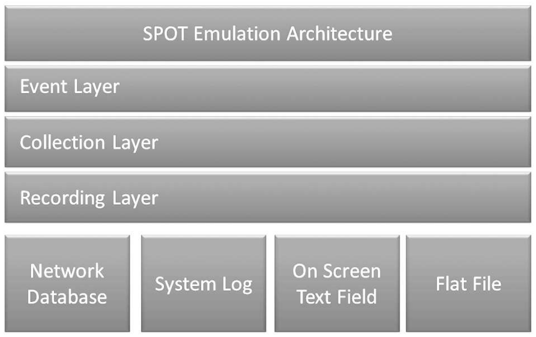


Fig. 6: SPOT Instrumentation System

SPOT uses an event-driven architecture that precisely identifies the occurrence of each major application-state change, such as registering or unregistering a GPS listener and SPOT takes a “snapshot” of the power consumption when the application performs these operations. This event-driven architecture allows developers to understand the power consumption of the application before, during, and after key power-intensive components.

In addition to event-triggered power snapshots, SPOT also periodically collects power consumption information. This information allows developers to trace overall power consumption or power consumption within a given block. The power information `Collector` that collects snapshots and periodic samples can be configured to run in a separate process to prevent contamination of the data.

To accomplish event-driven power profiling, SPOT fires events immediately before an application enters a component that was defined in the model and immediately after it leaves a model-defined component. These events work in conjunction with the periodic power consumption updates to provide developers with a complete description of how their software architecture elements consume power. SPOT’s event-driven model of collecting power consumption data also allows developers to identify precisely what the application was doing when key power consumption spikes occur, further helping them optimize their designs.

SPOT’s emulation infrastructure currently runs on the Android mobile platform and uses the power consumption API utilized by the “FuelGauge” application in the core Android installation. The power consumption API provides application developers with access to the amount of time the application utilizes the CPU, sensors, wake-locks, and other system resources, in addition to the overall power consumption.

Android’s power consumption API provides power-consumption information on a per-package basis. By implementing SPOT in a different package, developers can analyze power consumption without influencing it. Collecting power consumption information in this manner increases accuracy. Moreover, SPOT can be implemented simply as a collector to analyze existing applications without modifying their source code.

5.4 Pros and Cons of SPOT's MDE-based Approach

SPOT's focus on MDE provides a number of capabilities and restrictions on the breadth and accuracy of what can be modeled. A key benefit of SPOT's MDE-based approach is that the models are easy to construct and generation of emulating mobile application code does not require any manual development effort. This approach enables early and low-cost analysis of the impact of architectural decisions on power consumption. Moreover, SPOT's domain-specific modeling language (SPOML) shields developers from lower-level details that are not relevant to optimizing power consumption.

The key downside of SPOT's MDE-based approach, however, is that SPOML cannot express every possible application behavior and thus may not provide perfect fidelity to a mobile application that is eventually implemented. For example, an application that provides alerts based on complex calculations involving the accelerometer and GPS may not be expressible in SPOML; which would make it hard to reason about how the algorithm itself affects the power consumption profile of the app. For these types of scenarios, SPOT allows the integration of hand-coded application components into an application to provide greater fidelity. The downside of this flexibility, of course, is that components must be manually implemented if they are not expressible in SPOML.

6 Results

This section analyzes the results of experiments that empirically evaluate SPOT's MDE-based capabilities presented in Section 5. These experiments measure SPOT's ability to collect power consumption information on a given model, as well as accurately model the power consumption of a proposed application software architecture. These results show how SPOT can assess and analyze power consumption information gathered through the Android's power consumption API and evaluate SPOT's accuracy in predicting power consumption information about a software architecture at design time.

6.1 Hardware/Software Testbed

All tests were performed on a Google Nexus One with a 1Ghz CPU, 512MB of RAM, 512MB of ROM and a 4 GB SD card running the default installation of Android 2.1 (Eclair). The SPOT application was the only third-party application running at the time of experimentation. The same power consumption information gathering logic was used to collect information on emulation code, as well as the sample applications. The information was analyzed in Excel based on power consumption data written to the device's SD card in the form of a CSV file.

To assess the consumption characteristics of different designs, the current SPOT version generates an Android application package. It then periodically samples the battery usage statistics from the OS writing these values to a CSV file on the SD card. SPOT also fires events when the application's state changes, *e.g.*, when the GPS is started or a sensor is disconnected. These events allow SPOT users to examine the power consumption of active hardware, in addition to the overall consumption of the application.

SPOT uses an XML-based configuration file that is generated from the SPOML model described in Section 5.1. This XML file is loaded on to the device's SD card and parsed at startup. Due to the way that the power consumption API collects information, the data gathered reflects only power consumed by the SPOT application and does not include any power consumed by system processes, such as the GUI display or garbage collector.

6.2 Experiment 1: Empirical Evaluation of SPOT's Emulation Code Accuracy

Overview. This experiment quantitatively compares the power consumption of two Android applications and the power consumption of the emulation code derived from their respective SPOT models. Ensuring SPOT's accuracy is important since it allows developers to compare the power consumption of key power consuming components in their mobile architecture.

The applications used in this experiment are the WreckWatch application presented in Section 3 and OpenGPSTracker (`open-gpstracker.googlecode.com`), which is an open-source Android application that uses GPS to track the coordinates of the phone and display it on a Google Map on the device. The GPS points, and other information about the route, are stored on the device to allow the user to replay the route later. OpenGPSTracker also determines device speed as GPS points are collected.

Hypothesis. SPOT is intended to provide developers with an estimate of how a proposed application software architecture will consume power. We therefore hypothesized that SPOT could provide power consumption information to within 25% of the actual power consumption of WreckWatch and OpenGPSTracker. Based on prior work [25, 29, 26], we also hypothesized that the components we chose represented the key factors in mobile application power consumption and would be adequate to provide this level of accuracy.

WreckWatch results. Figure 7 shows the graph of the actual power consumption of the WreckWatch application compared with the power consumption of the WreckWatch emulation code generated by SPOT. The emulation code's power consumption follows the same trend as that of the application and provided a final power consumption value that was within 3% of the actual power consumed by WreckWatch. The SPOT model consisted solely of GPS and accelerometer consumers and did not attempt to model any additional aspects of the WreckWatch application. The model was accurate due to the substantial amount of power required by the GPS receiver. This result confirms that the GPS, sensor, and network modules that SPOT provides are key determinants of mobile power consumption. Although WreckWatch performs a number of CPU-intensive calculations on sensor data to determine if an accident occurred, these calculations are minor power consumers compared to sensor users.

OpenGPSTracker results. Figure 8 shows the graph of the actual power consumption of the OpenGPSTracker application compared with the power consumption of the emulation code generated by SPOT. As with the WreckWatch emulation code, the OpenGPSTracker emulation code consumes power at a rate similar to the actual application. Over the same time period, the emulation code for the OpenGPSTracker application was accurate at predicting power consumption to within 4%. The SPOT model for the OpenGPSTracker application only used a GPS consumer and did not attempt to model any Google Maps functionality (or requisite network functionality) or any processing required to determine speed or store the location information. In this

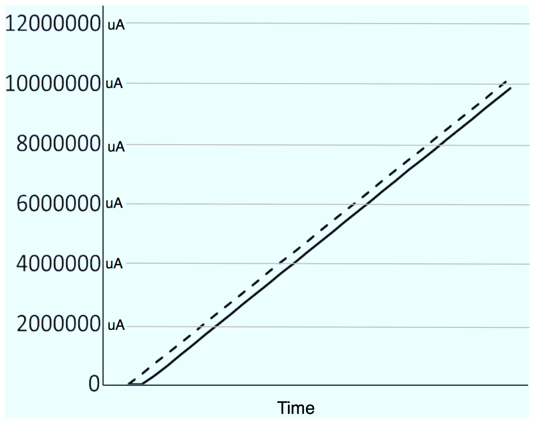


Fig. 7: Comparison of WreckWatch Application Logic and Emulation Code

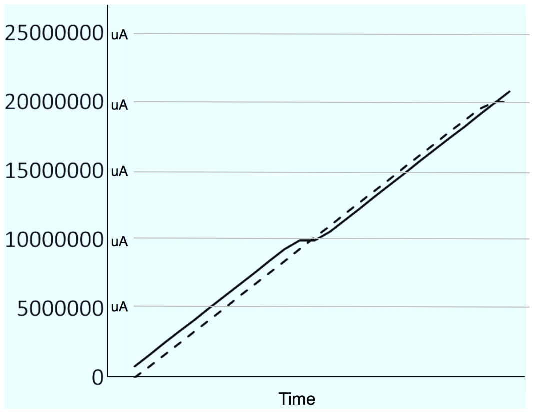


Fig. 8: Comparison of OpenGPSTracker Application Logic and Emulation Code

instance, the GPS power consumption was sufficient to model the power consumption of the entire application.

With both applications, SPOT modeled and predicted the power consumption of the actual application to within 4% of the actual power consumption. This result confirms our hypothesis that SPOT can provide an accurate prediction of power consumption by modeling key components of a mobile application.

6.3 Experiment 2: Evaluating the Accuracy of Simplified Architectural Models

Overview. This experiment shows that modeling the power consumption of the largest power consumer can be sufficient to accurately estimate power consumption, even though there are significant differences in the power consumed by the various sensors, CPU operations, and networking components of an application. SPOT provides a limited set of key power consumption elements for modeling a mobile software architecture. Although there is a finite set of elements for modeling power consumption in SPOT, a small number of these elements can often accurately determine the power consumption of a variety of applications, *e.g.*, modeling the power consumption of the GPS device is nearly as accurate as modeling the power consumption of both the accelerometer, CPU, and GPS of an application because the GPS receiver consumes so much power.

For experiment 2, we modeled the GPS sensor utilization of an app that used two sensors: the GPS and the accelerometer. The GPS was configured to sample every 500 milliseconds while the accelerometer was configured to use the FASTEST Android sample rate constant. We compared the power consumption of the emulation code, which only used GPS, to the actual app that used both the GPS receiver and the accelerometer. Modeling the largest power consumers allows users to model their designs even earlier by allowing them to ignore implementation details during the modeling process.

Hypothesis. We hypothesized that certain hardware components, such as GPS, consume so much power that emulating them with SPOT will consume nearly the same amount of power as the actual application. For example, an application that uses a GPS sample frequency of one sample per 500 milliseconds will have roughly the same power consumption as another application using the sampling the GPS every 500 milliseconds and receiving accelerometer updates.

Results. Figure 9 shows the results of experiment 2. As shown in the figure, the callouts represent the times that correspond with the entrance and exit from different components of the software architecture. This graph shows that the accelerometer had little effect on the overall power consumption in relation to the GPS. Despite running for 2.5 minutes after the deactivation of the GPS, the power consumption increases only a small amount in relation to the total power consumed by the application.

Figure 10 shows the relative power consumption of the several Google Nexus One components, such as the screen, accelerometer, Bluetooth radio, Wi-Fi module, GPS receiver and the CPU per time unit. As shown in the figure, certain hardware components consume a significantly larger amount of power than others. Moreover, the usage characteristics of these hardware components increase the difference in power consumption. For example, despite consuming almost as much power as the GPS, usually the Wi-Fi or Bluetooth is only active for short bursts of time whereas the GPS is usually left in

Fig. 9: GPS Power Consumption

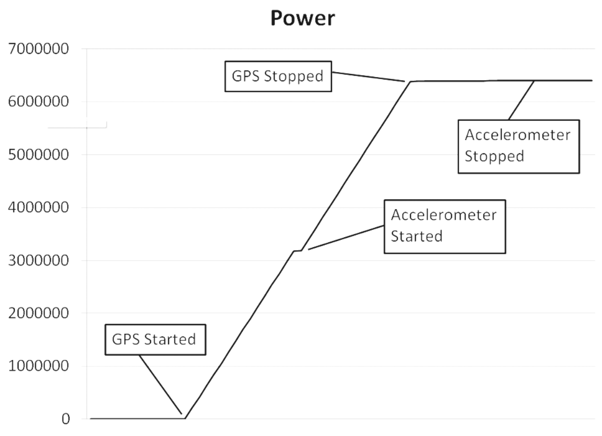
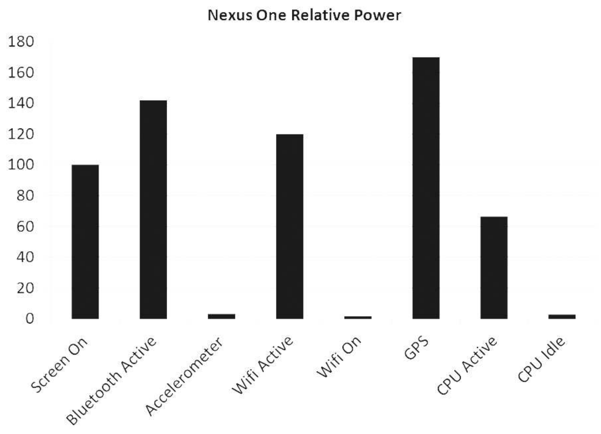


Fig. 10: Relative Power of Hardware Components



the active state for much longer periods of time. Due to these usage characteristics, the slightly greater power consumption per time unit of the GPS allows the GPS to quickly become the largest consumer of power on the device.

The results shown in Figures 9 and 10 support our hypothesis that mobile application power consumption depend largely on a handful of components, so SPOT can accurately predict mobile application power consumption by modeling these key power-intensive components.

6.4 Experiment 3: Qualitative Comparison of Sensor Sample Rates

Overview. This experiment evaluates the effects of sensor sample rates on an application's overall power consumption. The rate at which sensor data is consumed can have a significant impact on application power consumption, as described in Section 4.2. For example, Android's accelerometer provides four enumerations for sample rate: NORMAL, UI, GAME, and FASTEST. Although these sample rates provide varying levels of QoS, the trade-off between a given level of QoS and power consumption is not readily apparent at design time. The enumeration names give developers a clue to potential uses, as well as rank these sample rates according to rate and consequently power consumption. Alternatively, the GPS receiver allows developers to specify a value as the delay, in milliseconds, between samples.

SPOT allows developers to evaluate the power consumption of potential sensor sample rates. For experiment 3, we compared the power consumption of the GPS receiver while sampling at once every 2 seconds, once every 30 seconds, and once every 500 milliseconds.

Hypothesis. We hypothesized that SPOT could capture the power consumption effects of sensor sampling rate changes. In particular, we believed sampling rate changes of a few hundred milliseconds would produce power consumption changes that SPOT could detect.

Results. Figure 11 show SPOT's output for three different sample rates for the GPS sensor. The dashed line represents the power consumption of the application when the

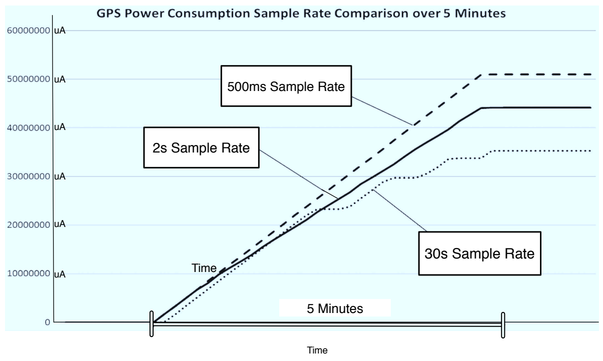


Fig. 11: GPS Sample Rate Comparison

sensor was sampled every 500 milliseconds, the solid line represents a sample taken every 2 seconds, and the dotted line represents the power consumption of the application sampling the sensor twice per minute. The samples in this graph were collected over 5 minutes and support the following observations:

• **Power consumption during the first several seconds is uniform regardless of sample rate.** Each graph is approximately equivalent for the first several seconds of data gathered during the GPS sampling, which implies that if developers need access to GPS for a short period of time, the greatest benefit would come from using a higher sample rate.

• **The greatest improvement in power consumption due to a lower sample rate will occur over time.** Although the graphs demonstrate a noticeable difference in power consumption over the 5-minute sample period, the improvement in battery life from a change in sample rate will only be realized when sampling occurs over an extended period of time.

Ultimately, the amount of power consumed by the GPS receiver is directly proportional to the sampling rate of the application. Reducing the sampling rate of the GPS receiver is an effective means to reduce the overall power consumption of an application if the receiver is active for longer than $\sim$2 minutes.

6.5 Summary and Analysis of Results

The results of the two experiments presented above show how SPOT can accurately analyze and predict an application's power consumption based on a model of the application software architecture. This capability allows developers to understand the implications of their design decisions early in the software lifecycle, *i.e.*, without implementing the complete application. The emulation code SPOT generates is accurate, *e.g.*, for our WreckWatch application it could predict power consumption within 3% of the actual application's power consumption. SPOT's accuracy stems in part from the significant power consumption of hardware components, such as the GPS receiver, that consume significantly more power than other hardware components on the device, such as the CPU or even accelerometers.

7 Concluding Remarks

The *System Power Optimization Tool* (SPOT) is an MDE tool that allows developers to evaluate the power consumption of potential mobile application architectures early in the software lifecycle, *e.g.*, at design time. Our experiments indicate that SPOT provides a high degree of accuracy, *e.g.*, it predicted the power consumption of the WreckWatch and OpenGPSTracker applications to within $\sim$3-4%. We learned the following lessons developing and evaluating SPOT:

• **Sensor sample rates play an important role in long-term power consumption.** The power consumed by the device sensors is largely uniform over the first several minutes of activation regardless of sample rate. It is only when these sensors are used for an extended period that the benefit of lower sample rates is realized. Developers must therefore consider the amount of time to activate the sensor in addition to the overall sample rate.

• **Certain hardware components draw significantly more power than others.** In the case of utilizing the GPS receiver, the power consumed by the location (GPS) service is so substantial that it becomes difficult to identify the effects of enabling or disabling other hardware components. Due to this "masking" effect, developers may overlook a significant consumer of power in an application. In general, small changes in GPS sample rate can have significant impacts on overall application power consumption.

• **Power consumption of an application can be accurately modeled by a few key components.** The power consumed by certain components, such as the GPS receiver and accelerometers is so significantly greater than the power consumed by running the CPU, displaying images on the screen, etc. that SPOT can effectively model the power consumption of an application simply by modeling those components. The most effective power consumption optimization can be realized, therefore, with changes to only a small number of components.

• **Certain system-related operations such as garbage collection are not reflected in data gathered by SPOT.** Through the current method of data collection SPOT is only able to gather power consumption information about operations that it performs such as CPU, memory or sensor operations that it specifically requests. Our future work will therefore develop mechanisms for capturing the impact of these services on power consumption.

• **Power consumption of networking hardware is dependent on data transmitted which is often dependent on user interaction.** The power consumption of hardware, such as the Bluetooth or Wi-Fi interfaces, is dependent on the data transmitted and received on the device. This data is often generated at run-time by user interaction with the program. While it is effective for the developer to generate some sample data and provide it to SPOT, it would be more effective if developers could simply describe the data, *e.g.*, via a schema file. Our future work is extending SPOT so it can process a data schema file and generate data that is random, yet still valid to the application.

• **Although GPS is a major consumer of power, not all applications rely on GPS.** Although GPS is a major consumer of power on today's smartphone devices, it is still important to understand the power consumption of applications that do not use the GPS, even if their power consumption is less drastic. Our future work is therefore analyzing SPOT's accuracy with mobile applications (such as 3D games with acceleration-based controls, streaming video players, and audio recording/processing applications) that do not use GPS, such as 3D games, feed readers, and multimedia applications.

SPOT is available in open-source form at `syspower.googlecode.com`.

References

1. Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless wakeups revisited: energy management for voip over wi-fi smartphones. In *ACM MobiSys*, volume 7, 2007.
2. A. Anand, C. Manikopoulos, Q. Jones, and C. Borcea. A quantitative analysis of power consumption for location-aware applications on smart phones. In *Proceedings of the 2007 IEEE International Symposium on Industrial Electronics*, pages 1986–1991, 2007.

3. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, Reading, MA, 2003.
4. J. Chen, K. Sivalingam, P. Agrawal, and S. Kishore. A comparison of MAC protocols for wireless local networks based on battery power consumption. In *IEEE INFOCOM*, volume 1, pages 150–157, 1998.
5. G. Creus and M. Kuulusa. Optimizing Mobile Software with Built-in Power Profiling. *Mobile Phone Programming and its Application to Wireless Networking, F. Fitzek and F. Reichert, Eds. Springer*, 2007.
6. L. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *IEEE INFOCOM*, volume 3, pages 1548–1557. Citeseer, 2001.
7. W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, volume 8, page 8020. Citeseer, 2000.
8. R. Herrmann, P. Zappi, and T. Rosing. Context aware power management of mobile systems for sensing applications. In *2nd International Workshop on Mobile Sensing*, April 16 2012.
9. J. Hill, D. C. Schmidt, J. Slaby, and A. Porter. CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. In *Proceeedings of 15th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, Belfast, Northern Ireland, March 2008.
10. J. Kang, C. Park, S. Seo, M. Choi, and J. Hong. User-centric prediction for battery lifetime of mobile devices. In *Proceedings of the 11th Asia-Pacific Symposium on Network Operations and Management: Challenges for Next Generation Network Operations and Service Management*, pages 531–534. Springer, 2008.
11. M. Kim, J. Kong, and S. Chung. An online power estimation technique for multi-core smartphones with advanced display components. In *Consumer Electronics (ICCE), 2012 IEEE International Conference on*, pages 666–667. IEEE, 2012.
12. M. Kjasrgaard. Location-based services on mobile phones: Minimizing power consumption. *Pervasive Computing, IEEE*, 11(1):67–73, 2012.
13. R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. *Wireless Networks*, 11(1):135–148, 2005.
14. A. Krause, M. Ihmig, E. Rankin, D. Leong, S. Gupta, D. Siewiorek, A. Smailagic, M. Deisher, and U. Sengupta. Trading off prediction accuracy and power consumption for context-aware wearable computing. In *Proceedings of the Ninth IEEE International Symposium on Wearable Computers*, pages 20–26. IEEE Computer Society, 2005.
15. R. Kravets and P. Krishnan. Application-driven power management for mobile communication. *Wireless Networks*, 6(4):263–277, 2000.
16. O. Landsiedel, K. Wehrle, and S. Gotz. Accurate prediction of power consumption in sensor networks. In *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*. Citeseer, 2005.
17. A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, pages 44–51, 2001.
18. T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 256–269. ACM New York, NY, USA, 2004.
19. S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian. Integrated power management for video streaming to mobile handheld devices. In *Proceedings of the eleventh ACM international conference on Multimedia*, pages 582–591. ACM New York, NY, USA, 2003.

20. D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 233–44. Citeseer, 2002.
21. T. Pering, Y. Agarwal, R. Gupta, and R. Want. Coolspots: Reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *Proceedings of the 4th International Conference on Mobile systems, Applications and Services*, page 232. ACM, 2006.
22. D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
23. C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA, September 2001.
24. N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. 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.
25. C. Thompson, J. White, B. Dougherty, and D. Schmidt. Optimizing Mobile Application Performance with Model-Driven Engineering. In *Proceedings of the 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 2009.
26. H. Turner, J. White, C. Thompson, K. Zienkiewicz, S. Campbell, and D. Schmidt. Building Mobile Sensor Networks Using Smartphones and Web Services: Ramifications and Development Challenges. In Maria Manuela Cruz-Cunha and Fernando Moreira, editor, *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*. IGI Global, Hershey, PA, USA, 2009.
27. Q. Wang, M. Hempstead, and W. Yang. A realistic power consumption model for wireless sensor network devices. In *Proceedings of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2006.
28. Y. Wang, B. Krishnamachari, and M. Annavaram. Semi-markov state estimation and policy optimization for energy efficient mobile sensing. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2012 9th Annual IEEE Communications Society Conference on*, pages 533–541. IEEE, 2012.
29. J. White, S. Clarke, B. Dougherty, C. Thompson, and D. Schmidt. R&D Challenges and Solutions for Mobile Cyber-Physical Applications and Supporting Internet Services. *Springer Journal of Internet Services and Applications*, 1(1):45–56, 2010.
30. J. White, J. Hill, S. Tambe, J. Gray, A. Gokhale, and D. C. Schmidt. Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques. *IEEE Software Special Issue: Domain-Specific Languages and Modeling*, July/August 2009.
31. J. White, D. Schmidt, and S. Mulligan. The generic eclipse modeling system. In *Proceedings of the Model-Driven Development Tool Implementors Forum at TOOLS 2007*, 2007.
32. C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX Annual Technical Conference*, June 12-13 2012.