Empowering Developers to Estimate App Energy Consumption

Radhika Mittal[‡], Aman Kansal[†], Ranveer Chandra[†]
[‡]IIT Kharagpur, Kharagpur, India
radhikamittal.iitkgp@gmail.com
[†]Microsoft Research, Redmond, WA
{kansal,ranveer}@microsoft.com

ABSTRACT

Battery life is a critical performance and user experience metric on mobile devices. However, it is difficult for app developers to measure the energy used by their apps, and to explore how energy use might change with conditions that vary outside of the developer's control such as network congestion, choice of mobile operator, and user settings for screen brightness. We present an energy emulation tool that allows developers to estimate the energy use for their mobile apps on their development workstation itself. The proposed techniques scale the emulated resources including the processing speed and network characteristics to match the app behavior to that on a real mobile device. We also enable exploring multiple operating conditions that the developers cannot easily reproduce in their lab. The estimation of energy relies on power models for various components, and we also add new power models for components not modeled in prior works such as AMOLED displays. We also present a prototype implementation of this tool and evaluate it through comparisons with real device energy measurements.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*

General Terms

Algorithms, Experimentation, Measurement, Performance, Design

Keywords

energy efficiency, display power, developer tools

1. INTRODUCTION

Poorly written apps can sap 30 to 40% of a phone's battery [10]. Battery lifetime is a common cause of frustration in smartphone users. Several efforts in research and industry are investigating techniques to improve the battery life, such as through use of higher battery density, dedicated low power processors for offloading computations from the application processor, or even off-loading to the cloud. However, such platform layer improvements do not suffice by themselves since poorly written software can callously eat into the resultant extra battery juice.

A significant portion of the battery is used when the phone is actively used in interactive foreground apps. While OS designers spend significant effort in optimizing the battery impact of background OS services, the foreground apps are largely controlled by app developers. Very few tools are available for developers to improve the energy efficiency of their apps. In fact, most developers are not even aware of the amount of energy their app consumes under a typical usage pattern. While they could use a power meter to measure the energy draw for their app during test runs, such an approach is not used in practice because it is tedious and expensive. It is very difficult to perform these measurements for multiple devices, network conditions, user settings for screen brightness, and so on. Even if measurements are performed, they are biased to the network conditions at the developer's location. Measurements by themselves do not give any insight into how much each component (CPU, network, or display) contributed to battery drain, making it harder to focus the optimization effort.

One alternative for developers is to use the "battery use" tool on Android phones, the Nokia Energy Profiler (NEP) or power modeling tools such as eProf [15]. While these tools can circumvent the need for power metering equipment, they are still limited in exploring multiple configurations, user settings, or network conditions. The profiled behavior remains biased to conditions at the developer's location.

In this paper we present WattsOn, a system that allows a developer to *estimate the energy consumed by her app in the development environment itself*. WattsOn can (i) identify energy hungry segments during the app run, and (ii) determine which component (display, network or CPU) consumes the most energy. For instance, an developer may determine if the battery drain is dominated by the download of a high resolution image is consuming more energy, the app display using a white background, or the computation that the app is doing. The developer may compare the energy impact of design choices such as developing a portable browser based app that downloads most content on the fly or a native app that only downloads incremental updates.

Furthermore, WattsOn allows *what-if* analyses, to answer questions, such as: How much energy is consumed on a different phone model? How does energy consumption change if the user has a 2G or a 3G network? What if the brightness is set to high? or, What if the app is used under a low signal strength area?

Our current WattsOn prototype emulates the power consumption for the display, network, and CPU, since these are the dominant energy consumers on a smartphone, consuming between 800mW to 1500mW in their highest power states. Other components such

MobiCom'12, August 22–26, 2012, Istanbul, Turkey.

as the GPU (peaking to 250mW - 350mW) and A-GPS (160mW - 350mW, across various chip sets), while significant, account for a smaller fraction of the total power consumption for interactive foreground apps. These components can be added to our framework provided appropriate resource consumption counters, relevant power models, and emulation strategies are available.

Specifically, we make the following contributions.

First, we present the first system that can estimate an appŠs energy consumption in different operating conditions (carrier, signal strength, brightness) without requiring expensive lab equipment and explicit measurements with repeated runs for each operating condition.

Second, we enable energy to be profiled within the development environment without requiring a specific mobile device. To achieve this, we scale the development machine's resources including the CPU and the network to match the characteristics of a real phone. The app thus behaves as if on a real phone and its energy impact on all components, such as idle display power-on time while waiting for a download, is correctly emulated. The combination of power modeling and resource scaling required for WattsOn design implies that only those power models and scaling techniques can be applied that are mutually compatible. WattsOn allows the app to use external resources such as web services from the Internet. These requirements are in contrast to other energy emulators, such as Power TOSSIM [22], based on event driven simulation, where all communicating nodes are within the simulator. Hence techniques such as virtual clocks cannot be applied in WattsOn for resource scaling. Processor power models based on architecture specific performance counters are also not amenable to use with scaled resources.

Third, we expand the catalogue of power models available for mobile devices. Power models for many components have been established previously in [2,4,6,8,17,20] at varying levels of complexity and accuracy. We use previously proposed models for CPU and WiFi. For displays, while LCD and OLED displays have been modeled in detail [5, 25], prior work did not provide a model for AMOLED displays. Hence we develop a new power model for AMOLED. For cellular networks, while the power model exists [18], measurements for model parameters with varying signal strengths and operators were not available in published literature. We perform new measurements to fill in some of the missing data.

Finally, we have validated WattsOn with multiple applications, devices, network conditions, and carrier networks. Average energy error varied from 4% to 9% across the apps tested. The accuracy of energy estimation when compared with the variability in hardware energy measurement for the same task over multiple runs indicates that WattsOn can offer a better energy estimate by eliminating several variable factors such as background activity on the mobile device. We also show how the component energy break-down produced by WattsOn can help application designers.

While the concepts of power modeling and resource scaling have been developed before, to the best of our knowledge, this is the first work that investigates the suitable selection of modeling and scaling techniques for development-time mobile app energy emulation, and validates their use through fine grained hardware measurements.

2. WattsOn SYSTEM DESIGN

Most mobile development toolchains provide an emulator to assist in app development such as the Android Emulator, the iOS Simulator, or the Windows Phone Emulator. While the emulators do not accurately reproduce all mobile device characteristics, the low overhead of their use makes them very beneficial for a variety of tests. WattsOn extends existing emulators to estimate app battery consumption.

The two major techniques used in WattsOn design are power modeling and resource scaling.

Power Modeling: One way to measure app energy is to use power metering equipment [7]. However, requiring every app developer to install such equipment is an arduous ask. Also, measurements have variability due to differences in network conditions, background activities running on the phone, not all of which can disabled by the developer. For instance, the measured energy after an energy reducing change made by the developer may turn out to be higher than before the change, due to degraded network quality at the time of the new measurement. Further, the measurement does not separate out the impact of display, network, and CPU that can be important for the developer to make their optimization decisions. To overcome these limitations, WattsOn computes energy from the resource utilization counters using power models [2,4,6,8,17,20].

Resource Scaling: Resources consumed by the app on the developer's workstation are very different from those on the phone. This leads to two challenges. First, the resource counters measured on the developer workstation cannot be fed directly into the phone power models. Second, timing of events might be different when running the app on the emulator than on the phone. Network packets may arrive much faster on the development machine, causing the user clicks for the subsequent tasks to be occur sooner, drastically changing the time spent on application tasks. Resource scaling addresses these challenges.

2.1 Design Overview

A block diagram of WattsOn is shown in Figure 1. The leftmost blocks represent the measurement of real device power characteristics required for *power model generation*[1]. These measurements may be performed by the smartphone manufacturers, mobile OS platform developers, or even volunteers using automated modeling methods [6]. The mobile app developer simply downloads the appropriate models.

On the developer machine, the app code for the mobile device runs in a mobile device emulator. We insert *resource scaling* techniques between the emulator and the actual hardware. As the app is executed on the emulator, we monitor its resource consumption using *resource profiling* methods available on the development workstation. The resource consumption monitored on the scaled resources is used in the *energy calculation* block to estimate the app energy using power models.

We describe resource scaling and power modeling methods for each of the modeled components below.

2.2 Cellular Network (3G)

The cellular network interface [17] consumes significant energy and we emulate it as follows.

2.2.1 Resource Scaling

The goal of resource scaling is to obtain the network resource consumption of the app as if executed on a real cellular data link. Multiple scaling options may be considered:

Virtual Clock: Event driven simulators such as ns-3 and Power TOSSIM [22] simply record time in ticks and the ticks can be mapped to the real time for the network nodes of interest while they may run much faster (or slower) on the simulation workstation.

[1]Power models may be developed for all mobile devices of interest; the number of models required may be reduced by considering representative devices in various device classes with different screen sizes and cellular network types.

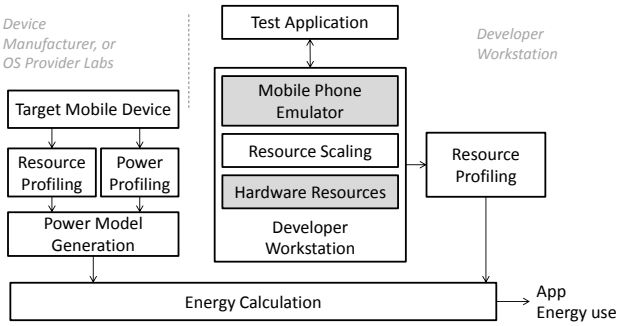

Figure 1: Block diagram of WattsOn components. Blocks shaded gray are used in existing systems; WattsOn adds the other blocks for energy emulation.

This technique is not suitable for WattsOn because the emulated app uses external resources such as web services on the Internet that are not operating on simulated tick time.

Trace Stretching: Another possibility is to capture the packet activity over the high speed network connection of the developer's machine and then stretch the timing characteristics to match those on a lower speed link. For example, suppose a 10 second trace would take 30 seconds over the cellular interface, one could multiply all time intervals in the Ethernet packet trace by 3. One could even replay the trace from the high speed interface on a lower speed interface simulated using a fine grained packet level simulator, such as QualNet or ns-3 and get more accurate stretching. However, this technique has several drawbacks. A slower interface will affect other parameters of the network flow, such as the TCP window size, which are not captured by stretching the high speed packet trace. Also, stretching the network trace alone will not produce the corresponding impact on other resources such as the timing of user clicks or other tasks in the app that depends on network activity.

Link Shaping: The approach we have taken is to shape the network link bandwidth and latency such that the emulated network activity in terms of packets sent and received is similar to the activity that would be observed on a cellular data link for the same task. We introduce our resource scaling code at layer 2.5 in the network stack, that is, between IP and the MAC layer. The delay, loss and bandwidth parameters are chosen to mimic different network conditions. The other components of the network remain the same as used by the app on a real device.

The latency, bandwidth and loss parameters for cellular networks have been studied in several prior works in wireless communications, and we rely on the existing literature to guide our choice of these parameters. Using measured distribution based models, rather than physical measurements, has the advantage that the estimate is much *more representative for a wide population of users*. It is not biased to the developer's specific phone location, and does not show uncontrolled fluctuations from measurement to measurement. Parameter values may be updates as technology evolves. In our prototype, we selected the parameter values as follows.

Latency: The measurement and modeling work in [9, 12] has characterized the cellular data link latency for 3G networks using a normal distribution (with mean = 200ms, and standard deviation = 100ms) and we use this characterization.

Bandwidth: We use the measurements from [23] that have experimentally characterized 3G HSDPA download and upload bandwidths. The bandwidth varies due to various reasons including changes in network congestion, wireless channel quality at the phone location, and other factors. To keep the number of varying conditions manageable, we bin the network quality into three levels, denoted *good, average*, and *poor*, and based on [23], set the parameter values as shown in Table 1.

Network quality	Download (kbps)	Upload (kbps)
Good	2500	1600
Average	1500	900
Poor	500	200

Table 1: Network scaling parameters for bandwidth.

Loss: We model losses using the well known Gilbert Elliot Channel Model and the corresponding parameters measured for 3G links from [26]. According to this model, the network is assumed to be in one of two states, denoted good and bad, each with a different loss rate. The model also describes the transition probabilities between the two states, that allows simulating the losses over time during an emulation run.

In certain instances, when the developer is using a slow Internet connection for their developer machine, such as a home Internet connection, then the underlying network may itself have non-negligible latency and losses. In this case, WattsOn should first probe and estimate the network latency using known methods [11] and then add on any additional resource scaling to the measured characteristics.

2.2.2 *Power Modeling*

The power model for the 3G network must model not only the active energy consumption when communicating data but also the "tail" time, or the time for which the radio interface remains in a higher power state after finishing the communication activity. Part of the tail time is spent in the active state (DCH) itself, and part in an intermediate state (FACH), where the radio power consumption is reduced but any further communication requires a small channel acquisition overhead. Some radios may have a second lower powered intermediate state (PCH). This model, called ARO model, was studied in depth in [17, 18], and allows back-calculating the radio power state from a network packet trace.

However, the number of mobile operators for which the model parameters have been measured is limited [18]. Second, the energy consumed at varying received signal strengths has not been reported for the ARO model, though dependence on signal strength is well known [21]. To fill in the above gaps, we set up two experiments (Figure 2).

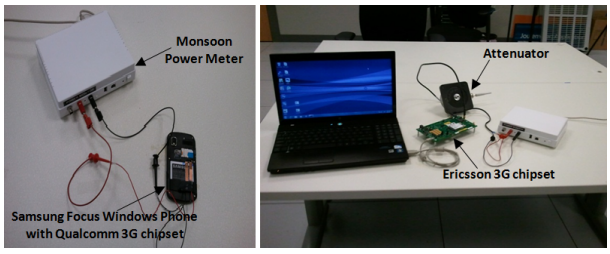

Figure 2: Experimental setups for the network power measurements.

Signal Strength. We used an Ericsson cellular data development board that exposes the antenna ports and allows controlling the received signal strength via an RF signal strength attenuator. The setup was located where we typically observed good signal strength. Measurements were performed on a weekend when the

network was lightly loaded. We varied the amount of data downloaded and uploaded at different signal strengths and measured the DCH and FACH power and tail times. Since a developer will likely only emulate their application with a small number of signal strength variations, we discretize the signal strength to three levels (Table 2). These power measurements are taken using a radio interface board and not a phone, implying that the idle power of the board can be different from that of the phone. However, the differences between the power levels at different signal strengths comes primarily from the radio and can be used to adapt the power model for varying signal strengths. The tail times for DCH and FACH did not vary with signal strength.

Signal Strength	DCH (mW)	FACH (mW)
High	600	300
Medium	800	300
Low	1500	400

Table 2: Cellular interface power variation with signal strength for the AT&T network.

Mobile Operators. The second setup uses a power meter attached to the battery terminals of a smartphone to measure power. We connected multiple devices from different mobile operators available in our region: AT&T (Samsung Focus), T-Mobile (HTC HD7), Verizon (HTC Trophy), and Sprint (HTC Arrive). T-Mobile and AT&T use GSM based networks with the UMTS standard from 3GPP for data. Verizon and Sprint use the Evolution-Data Optimized (EVDO) standard from the CDMA2000 family provided by 3GPP2. A sample power trace for a download using Sprint's network is shown in Figure 3. As expected, the radio stays in a high power state long after the data communication has completed. Similar measurements were performed for other operators and all measured tail times are listed in Table 3.

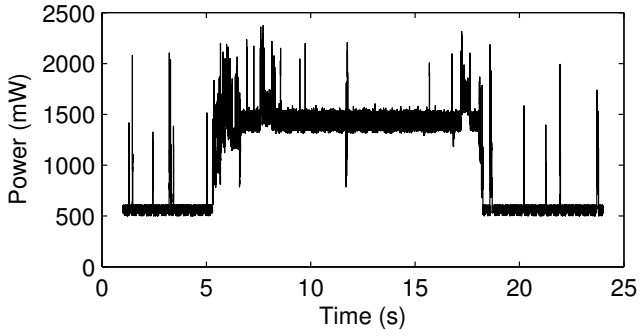

Figure 3: Network tail energy measurement for Sprint. The data communication ends near time = 7s along the x-axis but the radio stays in a higher power state for an additional 10 seconds after that.

2.3 WiFi Network

The resource scaling and power model methods for WiFi used are as follows.

Resource Scaling: For cases where the developer machine is connected to the Internet using WiFi (as is common for laptops), resource scaling is not needed. Otherwise, resource scaling is performed using the same layer 2.5 approach as for the cellular network. The scaling parameters are based on well-studied WiFi characteristics [1].

Operator	DCH	FACH	PCH
AT&T (3G)	5s	12s	0
T-Mobile (3G)	5s	1s	1s
T-Mobile (4G HSPA)	4s	2s	1s
Verizon (3G)	6s	0	0
Sprint (3G)	10s	0	0

Table 3: Tail state times for different operators. SOme operators do not use all intermediate states resulting in zero tail times in those states.

Power Model: The WiFi power model uses the PSM state model described in [13]. The model uses four states - Deep Sleep (10mW), Light Sleep (120mW), Idle (400mW), and High (600mW). The power consumption is slightly different for transmit vs. receive but since the difference is small and switching between these states is very frequent, we use a common power value, denoted as the high state.

When not communicating, the interface remains in Deep Sleep. Brief power spikes of 250mW at intervals of 100ms are observed in this state, corresponding to reception of beacons from an associated AP. When a packet is to be transmitted, the interface moves to the High state immediately. If a packet is to be received, the radio learns about it at the next beacon, and moves to the High power state. Once the transfer is completed, the radio moves to Idle. From Idle it can immediately move to High in case of a transmit or receive. If no network activity occurs in Idle state for 1s the radio moves to the Light Sleep. The Light Sleep tail time is 500ms, after which, if no network activity occurs, it falls back to Deep Sleep. If network activity does occur in either the Idle or Light Sleep state, the state changes to High. Both Idle and Light Sleep states also have regular spikes of additional 250mW of power spaced at 100ms to receive beacons.

The above power state transitions can be re-created, and energy can hence be computed, using the (scaled) network packet trace captured using a network sniffing library.

2.4 Display

As for other components we need to resource scale the display and model its power consumption. Fortunately, existing mobile device emulators already perform resource scaling for the display: the emulated app is provided only a small screen area representing the mobile device display. The only scaling needed is that the emulator window may be re-sized to a larger view by the developer, changing the number of pixels in the display and to overcome this, one may simply multiply the number of pixels by the appropriate scaling ratio.

While the peak power of the display is similar to that of the CPU and the network, the fraction of energy consumed by the display can be much larger than the other components since the display is constantly active throughout an application's use. An accurate estimation is thus important for this most dominant energy consumer. The power model for the display depends on the display technology used. Prior work has provided power models for LCD and OLED displays [5, 25]. However, several modern mobile devices use Active Matrix OLED (AMOLED) displays, that do not fit existing models.

The OLED power model shows linear and additive properties: the energy consumption of the display as a whole is the sum of the energy consumption of the R,G, and B components of the individual pixels. Further, the power model for the R,G, and B components is linear, provided the colors are converted from the standard RGB (sRGB) color space to linear RGB. This does not hold for

AMOLED. Figure 4 shows the power measured for the AMOLED display (on a Samsung Focus smartphone) set to different colors. The R,G and B components labeled are converted to linear RGB. While the linear model holds at low magnitudes, it breaks down at high magnitudes (the colors in the surface labeled B=255).

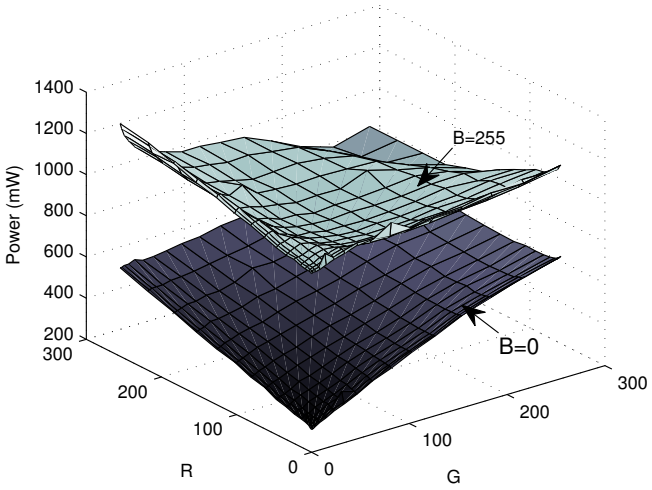

Figure 4: Power measurements for different colors on an AMOLED display.

Both the additive and linearity properties break down. As seen in the figure, power can increase or decrease with increasing color component values, implying that linearity does not hold. More measurements reveal that the power consumed is not just a function the color of a pixel but also depends on the properties of the other pixels in the image. Thus, the additivity does not hold either. Figure 5 shows the power consumption at varying fractions of the screen set to white. The graph shows the power predicted using the additive OLED model where the power would increase linearly with the area, as well as the observed ground truth. Power decreases when a greater portion of the screen is emitting brighter light levels.

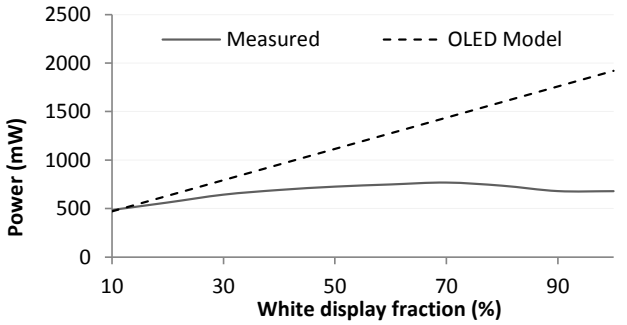

Figure 5: AMOLED power changes as the fraction of white colored pixels changes.

Further measurements reveal that reduction in power compared to the OLED model not only depends on the area but varies by color. Figure 6 shows an example measurement for two colors, compared with the OLED model based predicted values. While the predicted values are different for the two colors, what is important to note is that the difference between the measured and predicted values is also different for the two colors: $s2$ is significantly greater than $s1$.

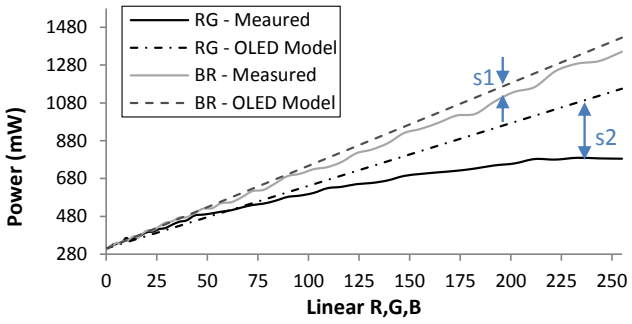

Figure 6: Change in power with color.

With display technologies being highly proprietary, the exact mechanisms used to control power are not widely available. Given the dependence on color and lack of explicit knowledge of the underlying power optimizations, the AMOLED power model becomes difficult to parameterize. Thus, we use a lookup table spanning the entire R,G,B gamut of values, discretized to 16 color magnitudes per component, yielding a table with $16 \times 16 \times 16 = 4096$ entries. The entries of this table contain the power value measured if the entire screen was set to that color, and in effect models the power scale down compared to the additive model at 100% area coverage for that color. The lookup table cannot be used directly however because the power value does not follow the lookup table when a color occupies less than the entire screen, and then its power consumption is closer to that predicted by the additive OLED model. Rather than fitting 4096 individual curves to model the area dependant power variation for the basis colors, we make a simplifying approximation that is based on the observation that the deviation from the OLED model is only significant for higher color magnitudes. The deviation does become significant at different threshold for different colors and we approximate that using four different thresholds: one for R+G+B magnitude, and one each for R+G, G+B, and B+R magnitudes, denoted $\tau_{RGB}, \tau_{RG}, \tau_{GB}$ and τ_{BR} respectively. The power scale down is assumed to be proportional to the fraction of such high intensity pixels.

The final model is summarized as follows. Suppose the lookup entry for a color indexed by $[r, g, b]$ by finding the nearest index in the lookup table is denoted $l(r, g, b)$. Let the power consumption of the entire screen, obtained by summing up the powers over all pixels of a screen-shot $\mathbf{s}$, be denoted $L(\mathbf{s})$. Suppose the fraction of pixels in $\mathbf{s}$ that exceed either of the thresholds $\tau_{RGB}, \tau_{RG}, \tau_{GB}$ and τ_{BR} is denoted $\beta(\mathbf{s})$. Also, suppose that the power consumption for the display using the additive OLED power model is computed as $O(\mathbf{s})$. Then the power consumption when the screen display matches $\mathbf{s}$ is computed as:

$$P_{display} = \beta(\mathbf{s}) * L(\mathbf{s}) + (1 - \beta(\mathbf{s})) * O(\mathbf{s}) \quad (1)$$

One further optimization to the above model is that instead of considering all pixel values on the screen, we could randomly select a smaller fraction of the pixels and use those as representative of the entire screen. The accuracy of this model along with sub-sampling is evaluated in Section 4.

From an implementation perspective, the pixel information of the display is easy to obtain when an app is being emulated. We simply capture the developer machine's screen and extract the region corresponding to the emulator's display area.

For completeness, we also measured the power consumption of LCD displays with varying screen sizes at two different color levels (Table 4). The LCD can be modeled using simply the brightness level as the variation with color is not significant. As expected, devices with different screen sizes do consume different amounts of energy.

Brightness	**Color**	3.6in	**4.3in**
Low	Black	132 mW	294 mW
	White	127 mW	330 mW
Medium	Black	363 mW	557 mW
	White	359 mW	573 mW
High	Black	559 mW	778 mW
	White	554 mW	790 mW

Table 4: Power measurements for LCD displays for HTC Arrive (3.6in screen size) and HTC HD7 (4.3in screen size).

2.5 CPU

Mobile devices use much lower power processors due to their battery constraints compared to the processors used on the developer workstations. The processor frequency, cache hierarchy, and various computational units are different, resulting in very different execution performance across the two processors for the same computation. Since processor designs have been extensively studied, many techniques for emulating performance and power have been developed for CPUs. We select resource scaling and power modeling methods that are mutually compatible and work with the full system app energy emulation constraints.

2.5.1 Resource Scaling

One approach to resource scaling would be to use a cycle-accurate simulation of the processor. Detailed cycle-accurate simulation has excessive computational overheads, and is impractical, especially since it may not run in real time and result in inaccuracies and delays in the timing of other components. The existing Android Emulator uses instruction set simulation through binary translation, built upon QEMU [16]. The Windows Phone Emulator similarly uses processor virtualization with support for memory and GPU emulation. While these lighter weight approaches followed by Android and Windows Phone emulators allow capturing certain CPU characteristics and memory limitations of the mobile device, they *do not preserve the timing characteristics* and do not model the processor architecture in sufficient detail to provide accurate resource counters. The Android Emulator allows for developer driven scaling of speed through a delay parameter that can take values between 0 and 1000 to add a non-deterministic delay to the application execution. However, the choice of the delay input is left to the developer.

In WattsOn, we scale down the performance of the emulated app running on the development machine by restricting the number of processor cycles available to the mobile device emulator. The goal is primarily to preserve the timing characteristics on the CPU. This is only an approximation because the nature of computation can affect how the number of cycles on one processor map to the other processor, given that the processor architectures are vastly different and aside from the number of cycles, the size of processor caches, accelerators and bus speeds will matter as well. We compare the execution time on a mobile device processor (Samsung Focus with a 1GHz Scorpion CPU) and a development machine using a 2.7GHz Intel Core-2 Quad-core processor for a few simply computational tasks including floating point computations, fixed point computations, and memory intensive compute tasks. The slow-down in execution time did not vary greatly across these tasks and for this pair of processors, the slow down was a factor of 7.2, implying that a 100% CPU utilization on the phone processor can be approximated using a $100/7.2 = 13.8\%$ utilization of one core on the developer machine processor[2]. The emulator can now be restricted to a fraction of the overall processor that yields the same slow down in execution speed. The fraction of restricted cycles is not exactly equal to the desired slow down ratio since the emulator may have additional overheads other than the execution of the test app[3].

The actual mechanism to restrict the number of cycles allocated to the emulator depends on the operating system used on the developer's workstation. For instance, in Linux, one may use the `cpulimit` utility. In Windows the same can be achieved via restricting the emulator to a virtual machine and restricting the CPU fraction allocated to it.

2.5.2 Power Modeling

Power models for the CPU are available in the literature [20] and we used a simple utilization based power model where the CPU power is expressed as a linear function of the phone's CPU utilization:

$$P_{cpu} = \alpha * u_{cpu} \quad (2)$$

Here, u_{cpu} represents the phone CPU utilization and α is a power model parameter. The CPU utilization measured by WattsOn for the emulator process on the developer machine is scaled according to the scaling factor in Section 2.5.1 to obtain u_{cpu}.

We measured the value of α on the Samsung Focus device for different computations. The above linear model being one of the simplest CPU power models is not perfect, and actual power consumption at the same peak utilization varied between 665mW to 781mW (a range of 16% w.r.t. the average) depending on type of computation performed. However, this model has the advantage that the scaling of the resource counter as described above works directly.

Other processor power models that are more accurate than equation (2) are available in the literature. While they can be used for run time energy profiling, their use in emulation is non-trivial because they require the use of multiple processor performance counters. Scaling all performance counters is challenging since most of the additional counters are highly dependent on processor architectures that vary a great deal between the developer workstation and the mobile device. For instance, the last level cache (LLC) miss counter is often found useful to improve power model accuracy. However, given that the cache hierarchy varies drastically between the two processors, scaling the LLC miss counter becomes impractical. Using the linear model from (2) enables a reasonable approximation at emulation time.

GPU: Some of the other components are also significant from an energy standpoint. The most notable one is the Graphics Processor (GPU). The GPU is mostly hardware managed and no software observable metrics regarding its utilization level are easily available. Power models for the GPU are thus not widely studied in the literature. Resource scaling for the GPU will also be non-trivial. We

[2]The scaling factor can easily be adapted for a new developer machine by measuring the execution time for a known processing task for which the phone size execution time is already known

[3]This approach assumes that the multi-threading characteristics of the app are preserved across the phone and workstation processors, i.e., if the app uses n threads on the phone, it will not use more than n threads on the developer workstation and the slow down is thus governed by the same bottleneck thread. The mobile device emulator generally ensures that this is indeed the case, especially if the instruction set of the phone processor is being simulated.

largely omit the GPU from our model. We measured the power consumption on a mobile device for several video playback tests and found that while some portion of the power measurement can be explained using the CPU and display, a significant portion remains attributable to the GPU: on an average 231mW across three different videos. Thus, if the emulated app is detected to be playing video, we can add this power value for the duration that the video is played. Other uses of the GPU such as graphics intensive games, are not covered by this approximation.

3. IMPLEMENTATION

Our implementation of WattsOn integrates the resource scaling and power modeling techniques described above, with the Windows Phone Emulator. Mobile emulators for other platforms such as th Android and iOS also execute on the developer workstation and most of our power models and network resource scaling techniques apply to them as well. However, Android uses a different CPU emulation strategy and preservation of timing characteristics may require additional changes.

WattsOn also adds a graphical user interface that allows the developers to explore multiple operating conditions for the run, such as network conditions (good, medium, bad), display brightness (low, medium, high), phone brand and model, network carrier (AT&T, Verizon, T-Mobile, or Sprint), and received signal strength. Since display brightness and signal strength do not affect the timing of the app execution, these parameters can even be changed after the test run and the energy estimates are updated on the fly using a previously obtained resource consumption trace.

The inputs to WattsOn include the power model parameters and the resource scaling parameters, encapsulated for a specific device into an XML file. Energy use across multiple devices may be emulated by simply changing the parameter file, without requiring access to a large number of physical devices.

The output of WattsOn includes: (i) a time series of power consumed for every component, and (ii) the total energy consumed. Using the battery capacity specification, we can also estimate how long the battery would last if a user continuously ran the app on her phone. A screenshot of WattsOn output is shown in Figure 7. The UI elements on the left allow the developer to change operating conditions while the 4 graphs show the energy used on the network, display, CPU, and the overall device respectively.

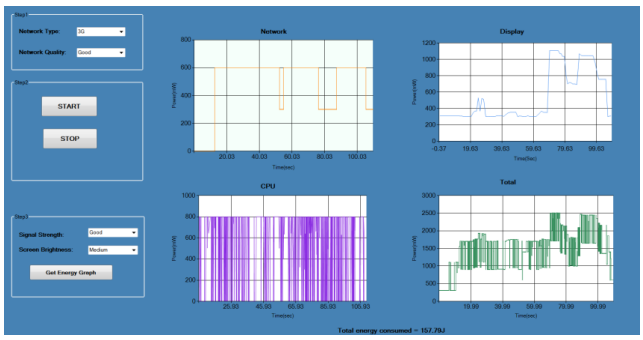

Figure 7: WattsOn output showing energy data. The UI elements on the left allow varying emulated operating conditions and technology characteristics.

While the prototype implementation includes power models for various environmental conditions such as multiple network signal strengths, user settings such as screen brightness, and also multiple hardware technologies such as LCD and AMOLED, additional configurations of test conditions are likely to emerge. The primary overhead of including any new configurations is that a one time measurement of that configuration must be performed with a hardware power meter, and provided for download to all developers using the tool.

4. PERFORMANCE EVALUATION

The accuracy of the energy estimate produced by WattsOn depends both on the correctness of resource scaling as well as the accuracy of the power models used. To evaluate the overall performance, we compare the estimated energy use as well as the measured energy use for multiple tasks performed on the mobile device. Since apps cannot be downloaded to the emulator from the app marketplace, we developed our own apps to perform these tests.

Application 1: Display only. We begin our evaluations with the display power model. While the LCD and OLED power models are available in prior work, the AMOLED model is new and we evaluate its accuracy using an application that consumed energy only in the display.

We first test the accuracy of our discretized lookup table that only stored 4096 colors out of the possible $255 \times 255 \times 255 = 16581375$, which is approximately only 0.024% of all possible color values. This test is performed with 100 random colors that are not in the lookup table, set to occupy 100% of the display area. This tests the accuracy of the modified AMOLED model for color dependent power scaling. Figure 8 shows both the estimated an the measured power and suffers from very little error.

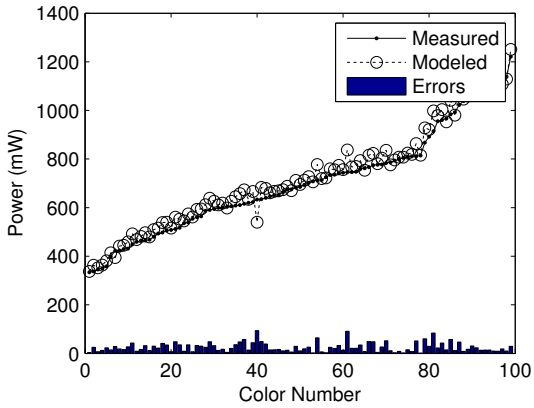

Figure 8: Testing the AMOLED display power model with 100 random colors.

For a more realistic test, we use an application that simply displays a static image and performs no CPU or network activity. Thirty different images encompassing various typical app displays including simple GUIs to rich textures and photographic imagery were included in this set. Figure 9 shows the measured and estimated energy (using equation (1)) for all 30 images. The energy estimated using 1% randomly selected pixels instead of considering all pixels is also shown. Again the model has very little error.

Another parameter worth exploring is the loss in accuracy suffered if we sub-sample only a small fraction of the pixels rather than recording all pixels. Sub-sampling can reduce the display tracing data overhead and may be important for long running tests. We report the incremental increase in error sub-sampling at 1% and 0.1% of the pixels in Figure 10. Reducing to 1% from 100% does

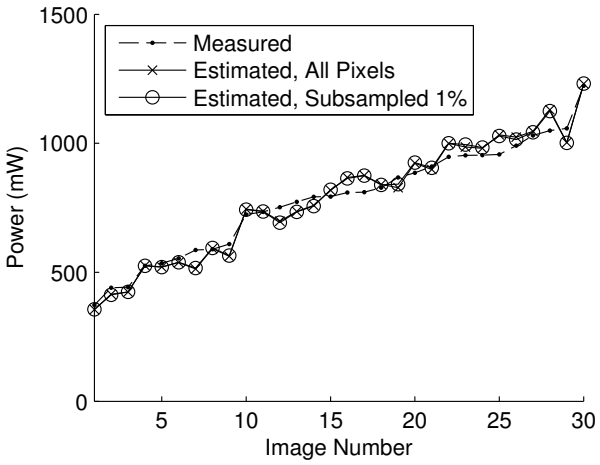

Figure 9: Measured and emulated energy for Application 1, with 30 different images. Images are sorted by the measured energy used.

not degrade the accuracy significantly (the gray bars indicating the loss in accuracy are very small).

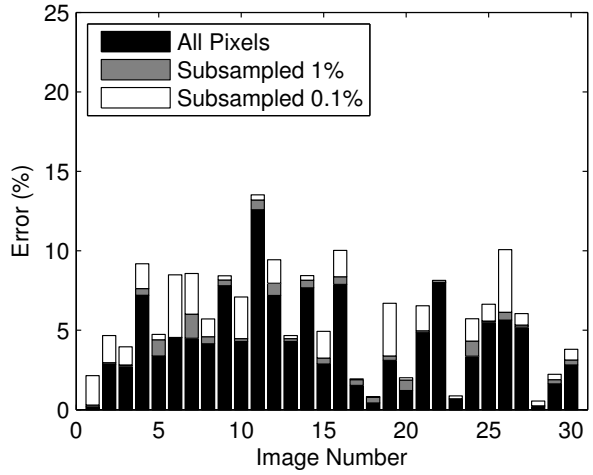

Figure 10: Display model error when all pixels are considered, or only 1% or 0.1% of the pixels are considered.

Application 2: Local Computation. This test is designed to model applications that use the processor and display, without network use or heavy graphics. The test app has a simple GUI, and user clicks perform computations executing over 5s to 20s. An example power trace captured using the power meter is showed in Figure 11(a). For the same task, the measured power trace averaged over 1s intervals as well as power emulated using WattsOn is shown in Figure 11(b). Power was measured with the mobile device in the airplane mode to avoid cellular network background activity. However, the device does have some background activity as seen in the raw trace near time 13s, and such activity, being independent of the app, should be excluded from consideration for the app developer. This is difficult in the measured power but is easy when using WattsOn.

Performing such an evaluation for the three different compute units, the total energy consumed over a 30s period, as measured

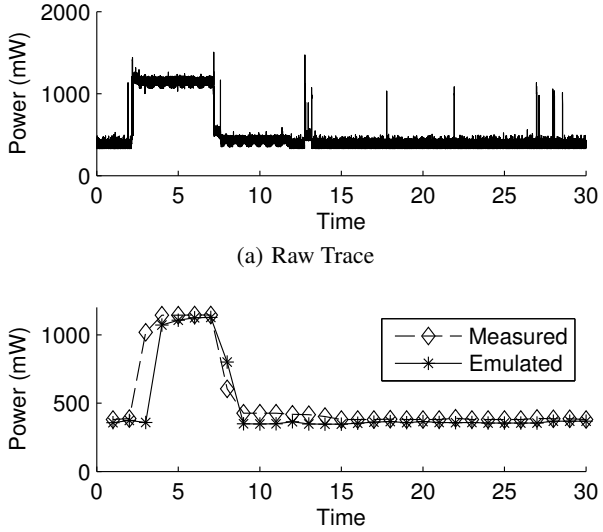

Figure 11: Comparing measured and emulated energy for Application 2.

and emulated is shown in Figure 12. Each measurement is averaged over 5 random runs and the standard deviations are shown as error bars. Since the resource scaling can behave differently from one run to another, the emulated energy can also vary slightly across runs. Overall, the emulated energy is lower since certain background activities are absent from the emulated trace. The mean error across all these runs is 9.3%. The primary factors leading to the error include any errors in the power model, the fact that background activities cause the measured energy to be artificially higher, and the error in resource scaling that may cause the emulated CPU usage to be different from measured.

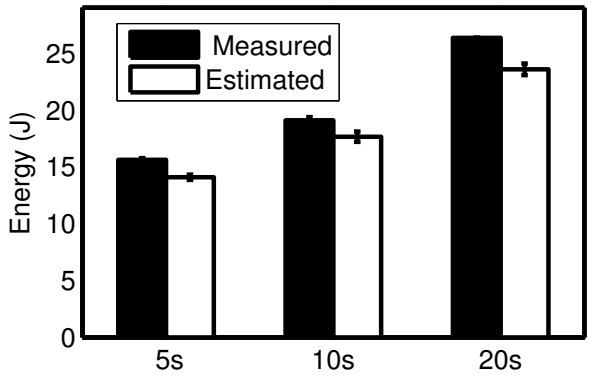

Figure 12: Experimental evaluation of emulation accuracy for Application 2.

Application 3: Networked Apps. Next we consider applications that use the network in addition to the CPU and display. A simple application is developed that can download files of different sizes from a predetermined web server. A sample power measurement is shown in Figure 13 and depicts the total power consumed by the display, network and CPU during one run. Performing the above test for different downloaded file sizes ranging from small to

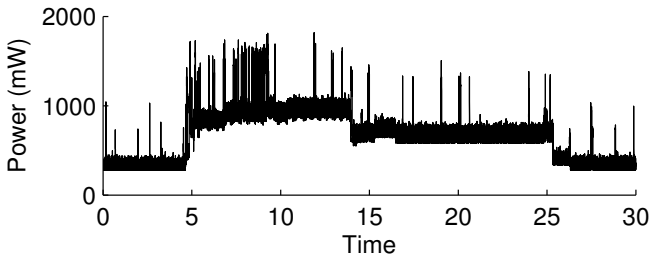

Figure 13: Example data capture for Application 3.

large, and repeating each test 5 times, the measured and estimated energy use is shown in Figure 14. The emulation energy is close enough for the developer to make the correct design choices based on the estimates provided by WattsOn. Average error is 4.73% across all tests.

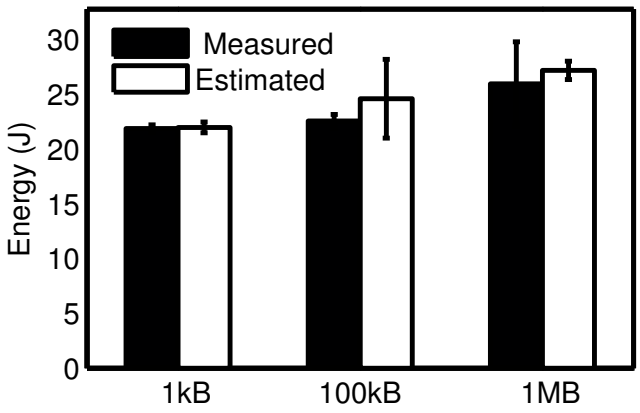

Figure 14: Estimated and measured energy for Application 3 with varying download sizes.

Application 4: Internet Browsing. This application downloads a webpage and renders it on the display. We test emulation accuracy over five different web pages (Table 5) differing in content size and complexity. The web pages are chosen to be static pages without variable advertisement content to make the experiment repeatable across a real device and WattsOn.

Web page	Size	Images
AMOLED Wiki Page	357kB	9
ACM	353kB	16
MSDN Mobile Apps	828kB	9
Google	117kB	4
NPS/Yose	532kB	14

Table 5: Webpages used for testing Application 4.

Energy is measured over a common session length of 50s for each URL. A variable portion of the session is spent on fetching the webpage for each case. There are variations across multiple runs due to network and web server variability. The measured and estimated energy is shown in Figure 15. The average error over all these experiments is 4.64% which is very similar to the variability in measurement one may observe even with real device measurements due to changing network conditions and device background activity.

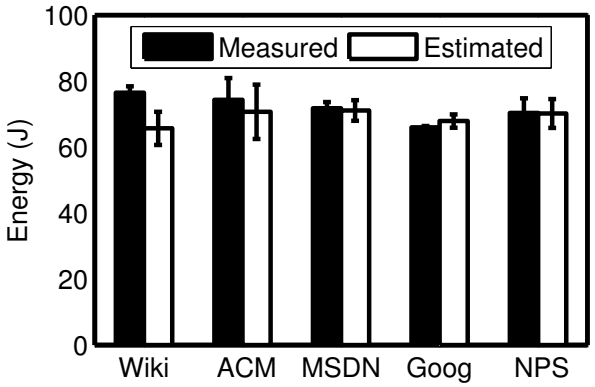

Figure 15: Accuracy evaluation for Application 4.

4.1 Case Study: Multi-Component Optimization

Let us consider an application that uses multiple components (CPU, network, display) and has design options to increase or reduce their use. A simple weather application suffices to demonstrate the trade-offs involved. While the application is simple, it allows illustrating how the break-down of energy use across multiple components provided by WattsOn helps understand the design choices more clearly.

The developer has multiple design decisions to make that may impact energy use:

1. **Portability:** The app may always download the entire content to be displayed from the web. Such application designs are sometimes referred to as hybrid apps and using the HTML5 standard, such apps can operate seamlessly on multiple platforms. Alternatively, the app may locally supply bulk of the content including weather images and only download succinct weather data from its web server. The latter approach reduces the amount of data downloaded from the Internet at run time but requires a platform specific app implementation on each mobile platform of interest.

2. **Rich Graphics:** The app may use different images for depicting the same weather condition: a simple cloud icon (such as Figure 16-(a)) only 18kB in file size, or a rich photographic cloud image which requires a larger 138kB file to be stored or downloaded (Figure 16-(b)). Richer graphics may allow for more sophisticated application UI designs but at the cost of higher data sizes.

3. **Animation:** The developer could even use an animated image of moving clouds to create a more engaging user experience, e.g., an animated image with two frames, one frame shown in Figure 16-(c), using a 90kB file. Animation would use up processor resources, and possibly require a larger image file than a simpler icon.

A quantitative energy cost estimate for each of the above three app features is necessary for the developer to determine whether to include them, and WattsOn can provide that with very low development overhead.

The decision depends on understanding energy consumption in multiple dimensions. Simply optimizing the app for lower CPU utilization will not necessarily reduce energy. Also, the display power model depends on color implying that even two images that

Figure 16: Sample images considered for an app.

appear similarly bright may consume different energy. Trying the multiple options in the app and recording the WattsOn output, while running the app for the same duration in each case, we obtain the data summarized in Figure 17.

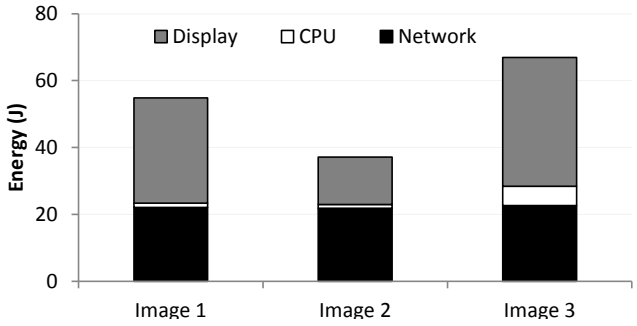

Figure 17: Energy breakdown into multiple components.

Several facts become obvious from this data. First, even though Image 1 is much simpler and has a smaller file size, it does not help save any energy on the network. This happens because for such small downloads the network energy is dominated by the tail states of the radio. A related implication is that saving the images locally and only downloading the text data related to weather will not help save much energy. Hence, including *rich graphics* and designing for *portability* has negligible energy overhead for this app.

Second, the display is consuming the largest fraction of the energy for this scenario. Clearly an image that saves energy on the display is likely to be beneficial, and other factors are less important to worry about during app energy optimization.

Third, looking in detail at the component breakdowns, one also notices that the CPU energy consumption of the third image is the highest. Looking at the emulation output from WattsOn (not shown for brevity) also shows that, while for the first two images the CPU energy is nearly zero, after the download completes, the CPU is consuming an average of 150mW continually while Image 3 (animation) is displayed. Since WattsOn shows the exact overhead of animation, the developer can determine if animation is important for their app scenario or not. Armed with such data, the developer can also decide if the app should highlight important information, such as a hazardous weather alert, using brighter colors or animation.

5. RELATED WORK

WattsOn builds upon a large body of work on energy modeling for phones [3, 6, 25]. The work in [3] focused specifically on a Palm device and measured several components including the CPU, LCD, device buttons, touch pen, and the serial link which was the primary communication channel on that device. On similar lines [25] is a more recent work that presents tools for automated execution of test benchmarks and measurement of power, to enable generation of power models for various components on the mobile device. The modeling approach was enhanced further in [6] to eliminate the need for external power measurement equipment, by using the battery drain measurements available on the mobile device itself. Aside from full system models, specific models for key components have also been studied in depth, including OLED displays [4], LCD displays [25], 3G cellular networks [17–19,21], and WiFi networks [19]. We leverage these established power modeling techniques in our design, and where needed, expand the set of models to cover additional technology variations.

Prior work has also looked at app energy accounting at run time. A key challenge for run time methods is the attribution of energy among multiple applications simultaneously using the device. In [24] resource usage of each component is tracked per application, and power models from the data sheet specs are used to estimate app energy. PowerScope [7] tracks the application with the active context on the processor (a single core CPU without hyper-threading was used) and measures the power at fine time resolution. The total power consumed when the application had active processor context is attributed to that application. Another powerful modeling approach, eProf [15], traces system calls made by applications and uses power state models for various components to infer energy used. It can incorporate tail-energy use, such as when a component remains in a higher energy state after the application is closed. We leverage similar power models but rather than profiling on a real mobile device we extend the work to enable energy emulation on the developer machine itself.

Energy emulation at development time has previously been developed in Power TOSSIM [22]. However, the event based simulation approach used in that work does not directly apply to mobile app emulation due to the interaction with external resources such as web services and interdependence among the timing characteristics of the network and other components. Different resource scaling techniques and compatible power models are hence required for WattsOn. Energy emulation methods using fine grained processor architecture models have also been developed [2]. However, such models are not available for all key mobile device components and have an extremely high resource overhead. Their use for full system emulation during app development with real time external interactions remains an open problem.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a system to estimate the energy consumption of apps during development. This enables app developers to use this feedback to write more energy efficient code. WattsOn scales down the emulation environment (network, CPU, display) to mimic the phone and applies empirically-derived power models to estimate app energy consumption. This approach gives us the flexibility to test an app's energy consumption under various scenarios and operating conditions. While we leverage known power modeling and resource scaling concepts, their combination for app energy emulation introduces certain constraints that requires a careful selection of compatible modeling and scaling techniques. We discussed such suitable techniques and experimentally validated their use.

We have prototyped WattsOn for the Windows Phone platform and shown its effectiveness for a variety of apps. Moving forward, WattsOn is just the first step in improving the energy efficiency of apps. Other useful steps include obtaining energy measurements from the wild to provide developer feedback based on real world usage patterns. Techniques to augment power models with real

measurement data to overcome modeling limitations [14] are also of interest. Together, we believe that these efforts will help developers produce apps that will significantly increase the battery lifetime of mobile phones.

7. REFERENCES

[1] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 209–222, 2010.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[3] T. L. Cignetti, K. Komarov, and C. S. Ellis. Energy estimation tools for the palm. In *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, MSWIM '00, pages 96–103, 2000.

[4] M. Dong, Y.-S. K. Choi, and L. Zhong. Power modeling of graphical user interfaces on oled displays. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 652–657, 2009.

[5] M. Dong and L. Zhong. Chameleon: a color-adaptive web browser for mobile oled displays. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 85–98, 2011.

[6] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 335–348, 2011.

[7] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, 1999.

[8] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 323–338, Berkeley, CA, USA, 2008. USENIX Association.

[9] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 165–178, 2010.

[10] S. Jha. Poorly written apps can sap 30 to 40% of a phone's juice., June 2011. CEO, Motorola Mobility, Bank of America Merrill Lynch 2011 Technology Conference.

[11] H. V. Madhyastha, T. Anderson, A. Krishnamurthy, N. Spring, and A. Venkataramani. A structural approach to latency prediction. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, 2006.

[12] J. Manweiler, S. Agarwal, M. Zhang, R. Roy Choudhury, and P. Bahl. Switchboard: a matchmaking system for multiplayer mobile games. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 71–84, 2011.

[13] J. Manweiler and R. Roy Choudhury. Avoiding the rush hours: Wifi energy management via traffic isolation. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 253–266, 2011.

[14] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX annual technical conference*, USENIXATC'11, pages 12–12, 2011.

[15] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 153–168, 2011.

[16] Qemu: open source processor emulator. `http://wiki.qemu.org/Main_Page`.

[17] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 321–334, 2011.

[18] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing radio resource allocation for 3g networks. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 137–150, 2010.

[19] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 255–270, 2010.

[20] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A comparison of high-level full-system power models. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 3–3, 2008.

[21] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 85–96, 2010.

[22] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 188–200, 2004.

[23] W. L. Tan, F. Lam, and W. C. Lau. An empirical study on the capacity and performance of 3g networks. *IEEE Transactions on Mobile Computing*, 7:737–750, June 2008.

[24] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 123–132, 2002.

[25] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, 2010.

[26] X. Zhao, Y. Dong, H. tao Zhao, Z. Hui, J. Li, and C. Sheng. A real-time congestion control mechanism for multimedia transmission over 3g wireless networks. In *Communication Technology (ICCT), 2010 12th IEEE International Conference on*, pages 1236 –1239, nov. 2010.