OS-based Resource Accounting for Asynchronous Resource Use in Mobile Systems

Farshad Ghanei
farshadg@buffalo.edu

Pranav Tipnis
pranavti@buffalo.edu

Kyle Marcus
kmarcus2@buffalo.edu

Karthik Dantu
kdantu@buffalo.edu

Steve Ko
stevko@buffalo.edu

Lukasz Ziarek
lziarek@buffalo.edu

Computer Science and Engineering
University at Buffalo, State University of New York
Buffalo, NY 14260-2500

ABSTRACT

One essential functionality of a modern operating system is to accurately account for the resource usage of the underlying hardware. This is especially important for computing systems that operate on battery power, since energy management requires accurately attributing resource uses to processes. However, components such as sensors, actuators and specialized network interfaces are often used in an asynchronous fashion, and makes it difficult to conduct accurate resource accounting. For example, a process that makes a request to a sensor may not be running on the processor for the full duration of the resource usage; and current mechanisms of resource accounting fail to provide accurate accounting for such asynchronous uses. This paper proposes a new mechanism to accurately account for the asynchronous usage of resources in mobile systems. Our insight is that by accurately relating the user requests with kernel requests to device and corresponding device responses, we can accurately attribute resource use to the requesting process. Our prototype implemented in Linux demonstrates that we can account for the usage of asynchronous resources such as GPS and WiFi accurately.

CCS Concepts

•**Software and its engineering** → **Software system structures;** *Power management; Real-time systems software;*

Keywords

Resource Management; Operating Systems; Embedded Systems; Power Management

1. INTRODUCTION

ISLPED '16, August 08-10, 2016, San Francisco Airport, CA, USA
© 2016 ACM. ISBN 978-1-4503-4185-1/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2934583.2934639

In the last two decades, computing has moved from desktops to mobile and embedded platforms. Advances in sensing, communication, and estimation algorithms have led to the development of advanced robotic systems such as driverless cars and micro-aerial vehicles. Networks of sensors reside in buildings as well as outdoor areas to monitor and improve our daily lives. Since such use cases rely on hardware that is battery powered, energy constraints lie at the heart of this evolution [16].

Energy is a finite, system-wide resource that needs to be efficiently managed across all applications. One requirement for this, is the ability to account energy usage for each application. This includes two steps. First to track energy usage, and next to associate the consumption to the application. This will allow another entity like OS to manage or police applications fairly. Traditionally, energy management mostly focused on the CPU and memory; however, current hardware is dominated by multitude of network interfaces, input modalities, sensors, and actuators. It is quite challenging to manage energy usage of these components, since it is difficult to accurately account for their energy usage. The primary reason for this difficulty is usage *asynchrony*, i.e., a resource can be used when the process that requested its use might not be executing. For example, getting a GPS fix occurs in the background; network packets are asynchronously sent from their processes since there could be multiple concurrent network transmissions. While modern frameworks such as Android track this type of usage in user space, there is no generic way to accurately determine resource usage at the OS level.

To address this challenge, we propose a general framework for accurate resource accounting in the OS. Our key observation is that *in order to reason about asynchrony, we need to relate a user-level request for a resource to the corresponding kernel-level request issued to the resource hardware*. We achieve tracking between the user space and the kernel with a general architecture that: (i) monitors the interaction between user processes and the kernel, (ii) monitors the interaction between the kernel and the hardware resources, and (iii) associates interactions crossing the boundaries between user, kernel, and hardware.

Contributions of this paper are (i) We identify asynchrony as the main challenge for accurate, system-wide resource accounting for modern computing systems such as mobile devices, wireless sensors, and robotic systems. (ii) We propose a general architecture that accurately tracks asynchronous uses of various devices. (iii) We implement a prototype of the architecture with two sub-systems (sensors and network interfaces) and demonstrate the benefits of our approach.

2. RELATED WORK

Although there have been many systems designed for accurate resource accounting, there is little research in handling the problem of asynchrony. For example, Resource Containers [1], the millywatt project [1] [18], Quanto [8], iCount [6], and ARO [15] all explore new abstractions and mechanisms for accurate resource accounting in various contexts. However, the support for asynchronous resources in these systems is either not considered or only considered for specific software systems (e.g., single-task operating system TinyOS in [8]). Further, the need for system-wide accounting of asynchronous resources has been highlighted as a challenge previously [17, 18].

One of the main applications of accurate resource accounting is energy management, which is an area of active research due to the proliferation of mobile devices. There are many approaches proposed in the past such as hardware-based mechanisms [10], energy consumption modeling [7, 2, 3, 4, 14, 13], and software-based mechanisms [11, 17, 14, 9]. FEPMA [9] is the closest to our work. It relates energy consumed to events registered by device drivers in the OS. However, they do not have the ability to attribute energy usage accurately (for asynchronous resources) to user applications, nor do they provide a general architecture that requires minimal implementation for future devices (and device drivers) to integrate into a generic accounting mechanism in the kernel.

3. RESOURCE ACCOUNTING

Modern computing systems such as robots, mobile devices, and wireless sensors consist of many hardware components that are not found in traditional desktop PCs. Such systems contain basic computing elements (e.g., CPUs and GPUs) and customary network components (e.g., WiFi and Bluetooth) along with sensors (e.g., accelerometers, gyroscopes, cameras, and laser range finders), input devices (e.g., touch screen, gestures, and voice), newer network components (e.g., 3G/4G, NFC, and Zigbee), and power sources (e.g., batteries). In such systems, we can classify system resource (computing, network, sensing, and actuating elements) usage into two categories:

- **Synchronous:** Resources that are used when the requesting process is running on the processor.
- **Asynchronous:** Resources whose use could occur even when the requesting process is not running on the processor.

Of particular interest is the asynchronous use of resources, which are much harder to account for with traditional OS constructs. There are several causes for asynchrony. These could be:

- **Sensing/Actuation Latency**: Some sensors and actuators require large amounts of time (order of seconds) to complete their action. Therefore, there is a large period of time between a request and response. During this time, the requesting process is typically blocking on the request. This could cause the OS to schedule another process, moving the requesting process to the background, e.g., GPS location fix when going from indoors to outdoors.
- **Resource Optimization**: It has long been hypothesized that in sensor systems, procrastination or delaying network communication prolongs battery life [5]. For example, delaying communication in a non-time-critical application allows each sensor node in a network to batch several data units into one transmission, thereby amortizing the fixed cost in communication such as duty cycling, channel polling, synchronization, etc. In practice, this is observed to be similar to sensing latency.
- **Resource Multiplexing**: Multiple applications can concurrently request the use of a resource. This is a common occurrence for a network interface on a mobile device. It is very challenging to ascertain the amount of resource use per application in such scenarios.

[1]http://www.cs.duke.edu/ari/millywatt/

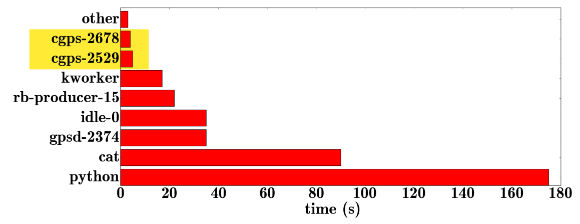

Figure 1: Cumulative scheduled time for each process

3.1 Accounting For Asynchrony

These causes of asynchrony make it challenging for user space applications and current OS mechanisms to reason about resource use. For example, accounting could be incorrect if we attribute resource use in a synchronous fashion. Figure 1 shows an example of GPS usage that we have measured, and the period of time that various processes were running on the processor during the GPS fix. The two processes requesting the resource (GPS) ran on the processor for a total of under 5%, while other processes not requesting their use ran for most of the time. If we attributed this resource use synchronously, 95% of the GPS use would be attributed incorrectly. Therefore, we need specific mechanisms to account for asynchronous resource use.

The key observation that helps us reason about asynchronous resource use is the need to *relate a user-level request for a resource (from a user space process to the kernel) to the corresponding kernel-level request for the resource (from the kernel to the hardware resource), and similarly the response from the resource to the kernel and the kernel to the user process.* This connects the application request to the time period that the device was in use, thereby providing a more accurate accounting of resource use.

3.2 Resource Accounting (RA) Architecture

Given these observations, our goal is to design a generic architecture to perform resource accounting of asynchronous resources. Our Resource Accounting (RA) architecture is intended to be re-usable, general, and easy to adopt, i.e., requiring minimal changes in the kernel source code from

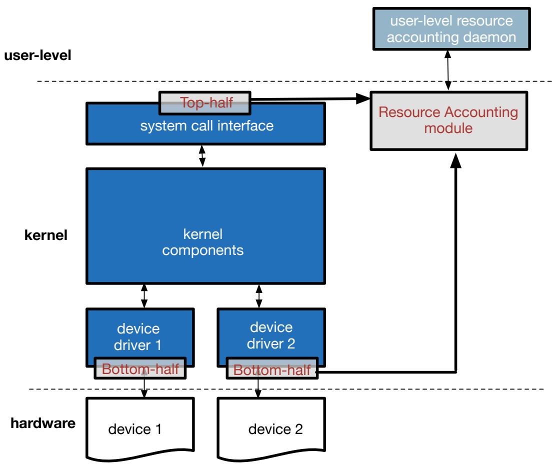

Figure 2: Our Resource Accounting architecture

the developers for current, and future devices. Our design in the kernel consists of three main components:

- Top-half: Instrumentation monitoring the user space-kernel boundary.
- Bottom-half: Instrumentation monitoring the kernel-hardware boundary.
- RA module: Bookkeeping module that associates interactions crossing the boundaries at the top-half and bottom-half in the kernel.

We illustrate our additions to two OS kernel sub-systems in Figure 2. In order to track a request through from the user process to the actual device call, and back, we need to observe the point of entry of the request from the user to the kernel and the kernel to the device, and relate the two occurrences.

The *top-half* instruments kernel system calls to intercept resource usage requests and identifies specific information about the request that helps us identify this request uniquely. The *bottom-half* instruments the kernel device driver portion that places the request on the use of a device. The bottom-half records information that helps us correlate it with the corresponding request from user space. This correlation is feasible in most cases, but challenging in some. We have found that the storage sub-system in Linux has multiple levels of transformation going from VFS to a file system (ext4 for example) to the buffer cache making it hard to relate a write call to the corresponding call to the buffer cache. We are currently working on additional instrumentation for this. However, we believe that the storage sub-system is the only system that is challenging to correlate. Finally, our design consists of a *Resource Accounting module* that performs the bookkeeping. Both the top-half and bottom-half record their observations with the RA module. The RA module provides this correlation information to user space through an interface so that the user processes can query their resource usage at runtime.

This architecture requires minimal additions to enable newer devices into our accounting. We provide instrumentation to the system calls (top-half) and also to the device drivers for a wireless card as well as a GPS device (bottom-half). Each new device will require a couple of methods implemented in the device driver for the bottom-half. There is a resource accounting user space daemon that reads the top-half and bottom-half interactions and accepts queries at runtime for application resource usage.

4. RA IMPLEMENTATION AND EVALUATION

We implement this architecture described above in Linux kernel 3.4.0 running on the DragonBoard APQ 8074. To demonstrate our design, we instrument two sub-systems of the Linux kernel to attribute resource usage of two classes of asynchronous resources - network (WiFi) and sensors (GPS). We will first describe the implementation, and the challenges involved in Section 4.1. We will evaluate the accuracy of our attribution for each of the components in Section 4.2. In Section 4.3, we demonstrate the utility of RA by running several applications concurrently, and attribute resource usage per application. And finally, we evaluate overhead of our accounting overhead in Section 4.4.

4.1 RA Implementation

4.1.1 Sensors (GPS)

The DragonBoard comes with an internal GPS that has binary drivers as well as user space processes that can be downloaded. However, this is not useful for our effort as we need to instrument the driver. To get around this problem, we connected the DragonBoard with an external USB GPS [2] that has known open source drivers.

Note: Generally for closed source drivers, a wrapper can be added in the kernel source code just before sending the request to the driver, and right after receiving it from the driver. This gets us as close to the hardware as possible, before entering the driver code.

Our sensor connects to the DragonBoard via a USB-serial interface. This interface is handled by the Linux kernel via the `tty` subsystem. The user processes interact with the device using the `open, close, read, write, ioctl` system calls. We instrument these for the top-half, or the time of the request by a user process. We also instrument the Linux `tty` subsystem to intercept the call before it is sent to the device driver. This is the bottom-half, or the time when the request is issued by the kernel to the device.

The GPS is queried by user space processes through a user space daemon `gpsd`, which interacts with the serial device. Applications intending to use the GPS connect through a socket to `gpsd` and request the GPS location. `gpsd` itself passes on the request to the GPS device via the USB-serial device and caches the location. The use of this daemon also presents an additional challenge. Our kernel-based accounting associates all requests to this daemon, and is unable to attribute it to the application that is actually making the request. To overcome this, we had to instrument the `gpsd` daemon to provide the port number of the connecting application to the user space RA daemon.

We define three states for our external GPS module, as it has been done in many applications: off, on, and active. When we connect the USB interface of the GPS module, it turns on and starts consuming energy. Since the GPS module is external and separate, we are able to measure the power being delivered to the module using a separate circuit. By comparing the so-called power used by solely

[2]The Adafruit Ultimate GPS Breakout based on the MTK3339 chipset.

the GPS module, with the overall system-wide power added by connecting the USB interface, we observe that connecting the USB interface also introduces extra power usage to the main board itself. Nevertheless, since that usage is also due to the usage of the GPS module, we include it into our *on* status.

4.1.2 Network (WiFi)

We also instrumented the network sub-system to demonstrate the generality of our resource accounting architecture. In particular, we use the WiFi network interface on the DragonBoard for this prototyping. We believe that these ideas should have a seamlessly transition to other network elements by just replicating the bottom-half in the particular network element's device driver. In Linux, users perform network interaction through sockets by invoking `send` and `receive` system calls. For the top-half, we instrument the `send` system call to timestamp the requests. Then to correlate it with the resultant kernel data structure (`sk_buff`) from that call, we maintain the port number as a proxy for the process that requested the use. However, multiple processes could concurrently use the network stack by using network sockets. This makes accounting for such a resource challenging.

Note: It is challenging to account for packet receptions with this architecture. This is because the resource is used before the resource usage is notified to the kernel - i.e., the WiFi card is active before the kernel is notified about packet reception. We are investigating other ways of incorporating this accounting such as per-component hardware instrumentation as done in [12].

A practical challenge in quantifying network resource use is that the WiFi module is on-board and inaccessible for power measurement separately. Therefore, we can only measure system-wide power. Thus we first characterized the rest of the system by running the board with no load, and for each of our experiments, we subtract the average steady-state power draw of the board to obtain the WiFi power draw.

We calibrated our resource accounting measurements by conducting several trials consisting of sending files of various sizes, recording the WiFi utilization, and attributing the energy consumed for that transaction across that utilization. By averaging across these trials, we got an average energy consumed per unit time of utilization. We use this for predicting energy consumption.

Turning the WiFi module on will add extra power for the module itself, scanning for APs, communicating, transfer of packets and so on. Since the purpose of this paper is not a detail characterization of network module, we only consider steady states for our model. Our contribution in this paper is not accurate modeling of WiFi energy draw, but accurate accounting of the modeled energy to each process. We use a simple WiFi power model that uses average power draw in various WiFi states. Our empirical observations also show that each transmission is followed by a short period (0.3s) of "tail power" draw before going back to idle. If another transmission is initiated within this period, the WiFi module will not go to idle mode until second transmission is finished. Modeling this state is important when multiple packets are being sent one after another. So we define this state as *active state*. This behavior is shown in Figure 3.

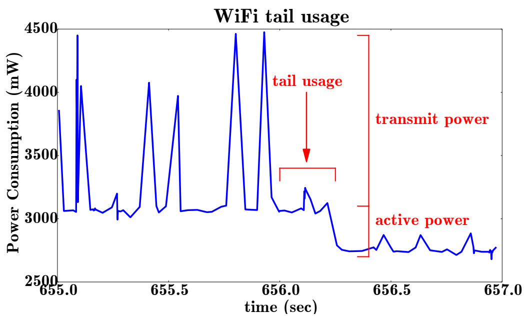

Figure 3: The *tail* usage of the WiFi module, after multiple consecutive transmissions with active state in between.

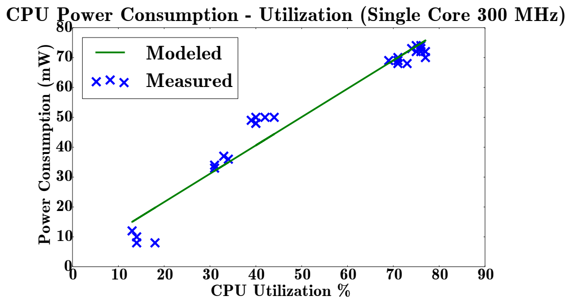

Figure 4: CPU power usage based on the total utilization.

4.2 Per-Component Resource Accounting Evaluation with RA

In order to accurately determine energy consumption of applications, we needed to characterize the energy consumption of the synchronous components as well. This includes the processor (CPU) and the rest of the system. We aggregate the rest of the system as one since we are not using other individual components of our development board.

By turning off all active components (GPS, WiFi, USB peripherals, screen), as well as putting the CPU in the lowest frequency and shutting down all the cores except one, we reach the lowest power usage by the board, which we call **board idle**. The base power usage of the board is consistent and by averaging over the results, we obtain 2430 mW as the base system usage when there is no activity.

Our development board has the Snapdragon 800 quad-core processor. To simplify measurements, we disable three cores, and put the one core running (core0) at the lowest frequency for tests. To measure CPU power draw at different loads, we ran several tests as shown in Figure 4. From

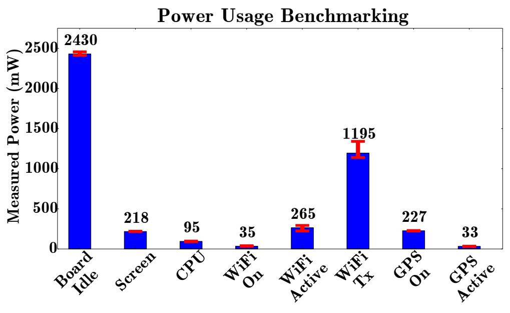

Figure 5: Measured power states of the system used to populate the power model.

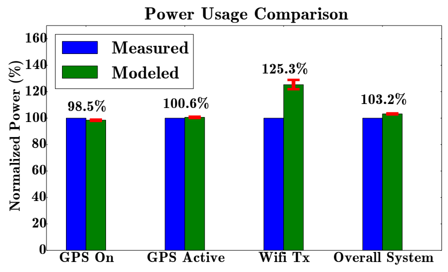

Figure 6: Testing our model against ground truth measurements.

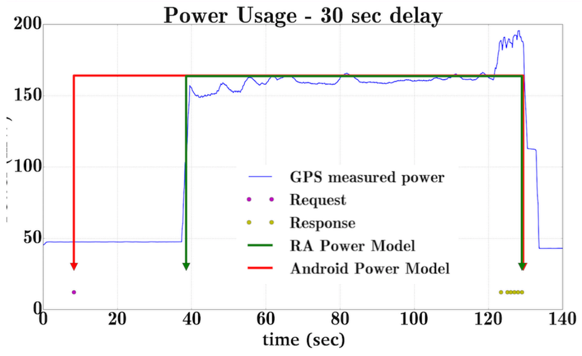

Figure 7: Sample GPS request with added delay to simulate optimization of use on the sensor.

our measurements, we fit a linear model for power draw of the CPU.

Then we ran several experiments to compare the individual components in our system, and their relative power draw. This is shown in Figure 5. Based on these numbers we build our model for RA-based energy estimation.

To understand the accuracy of RA estimation for asynchronous resource use, we compare average individual power draw from RA estimation to the measured ground truth by running experiments that utilize only those components respectively. This comparison is shown in Figure 6 for five trials with error bars. Our attribution is very accurate and within 2-3% for all except WiFi transmit. We believe that optimizations in the WiFi hardware might be the main cause for the 25% difference in estimated draw.

Note: As shown in Figure 5, a huge portion of system power consists of the board being on. Since this portion is relatively huge, and on the other hand very consistent, by considering this part our results will fall into 1-5% margin of the measured power in every test. Instead, we take that part away from the measured and modeled results, to be bale to compare the component-based portion of the power consumption.

4.2.1 Asynchrony Due to Resource Optimization

To simulate asynchrony caused due to request optimizations in the OS, we introduce a delay in the sensor request being forwarded to the sensor from the kernel. In Figure 7 (top), an application GPS request is delayed 5 seconds (in `gpsd`) before being passed on to the kernel. This models energy optimization similar to proposals in [5] and others. Since we correlate the timing between when the request is issued from user space, and when the actual request is passed

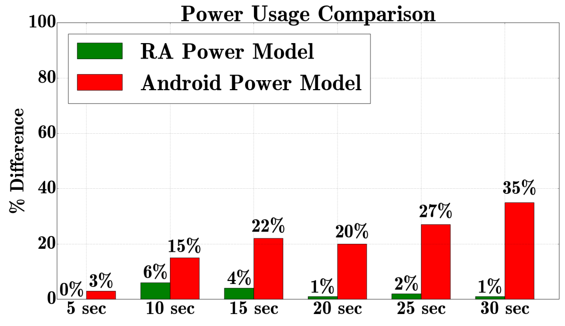

Figure 8: Error in energy accounting by accounting in user space (such as Android)

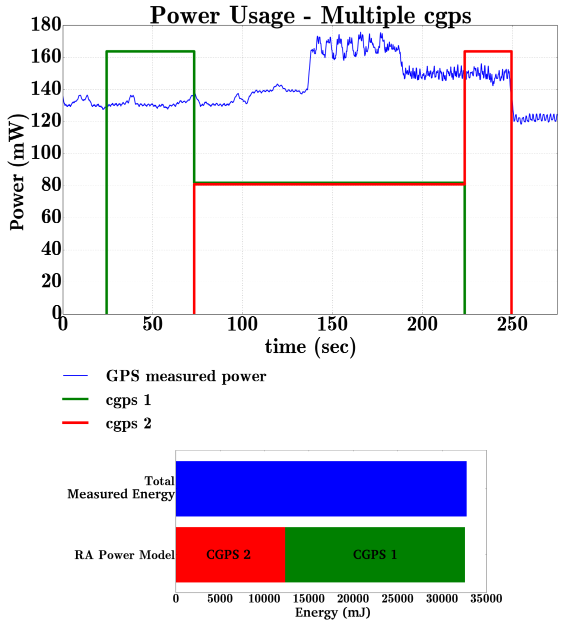

Figure 9: Two applications using GPS concurrently

on to the actual device, our RA architecture is able to discern such optimizations, both in user space and the kernel. Other mechanisms such as the energy accumulation in Android framework are done in the user space, and are unable to identify such optimizations, and are therefore inaccurate.

In order to demonstrate this benefit further, we ran several experiments varying the simulated sensor delay from 5s to 30s in increments of 5s. Figure 8 summarizes the energy attribution for each of these scenarios as a percentage increase or decrease from the measured power. Our resource accounting architecture performs quite well, and is only a constant factor off from the measured ground truth in comparison to accounting performed in user space.

4.2.2 Asynchrony Due to Concurrent Resource Use

The next challenge is accounting for multiple applications requesting the GPS use concurrently. Figure 9 shows a trial and the aggregated energy numbers with two `cgps` clients requesting a GPS fix concurrently. Our RA attribution divides predicted resource use across concurrent user processes. In this trial, our accounting comes within 1% of the measured energy use. We picked this trial at random, but it performed extremely accurately in this scenario. We believe that our

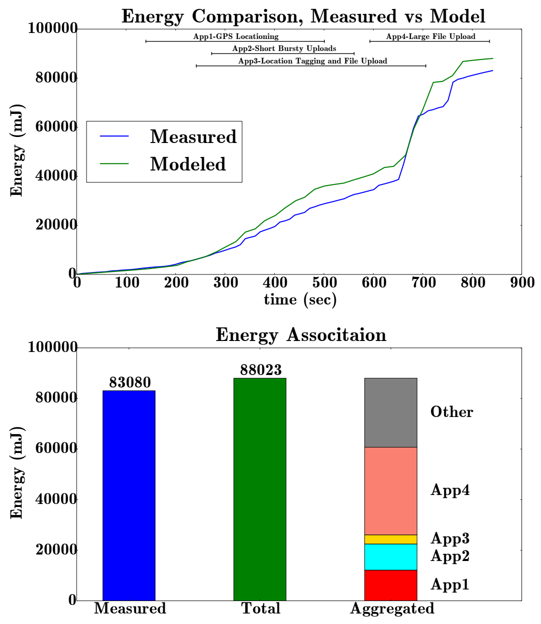

Figure 10: Per-app energy attribution for concurrent execution of multiple applications

prototype is quite accurate modulo modeling issues.

4.3 Per-App Resource Accounting

The previous section showed energy attribution done by RA for individual hardware components that cause asynchrony. Our RA daemon aggregates the resource usage of individual applications (processes), and records them as the energy consumed by that application for that execution. To test the efficacy of per-process resource attribution, we ran a script that executed several applications concurrently, varying CPU load with matrix multiplication, network transfer of files of varying sizes through ftp using WiFi, and requesting location from the GPS module. The energy attribution by RA per-app as well as in total can be seen in Figure 10, along with the ground truth measured energy. As can be seen, we are fairly accurate. Similar runs show that our attribution is within 10% of ground truth energy draw by the system for scripts running for 10-20 minutes.

4.4 RA Overhead

To validate our implementation and measure the overhead, we ran a long intensive mathematical program on the board that gets scheduled on the processor as much as it can. The program consists of a loop that repeats for 300000 iterations, multiplying two matrices of size 20x20. We ran this benchmark several times on the board's initial OS, and several times after adding RA modifications, to measure the average overhead added by RA. The test took an average of 714 seconds before applying RA modifications, and an average of 727 seconds after, which shows about only 1.8% reduction in performance.

5. CONCLUSIONS

In this paper, we have presented an accurate resource accounting approach for power constrained systems. Our approach monitors user-level and kernel-level resource requests and relates them, so we can accurately attribute a request and its actual resource use to its originating process. This approach addresses the difficulty of accurate resource accounting, especially for peripheral components such as sensors and specialized network interfaces.

Our evaluation shows that the approach is effective in accurately tracking the usage of asynchronous components when we use the time elapsed between a request and a response as a proxy for resource usage. With some careful modeling, we are fairly accurate in our estimation of energy use due to the resource. However, there are several challenges that we encountered performing accurate resource accounting—some of them are due to our choice of counting time as a measure of resource use, and others due to the nature of resource use. We discussed them in detail with the hope that the reader is better informed about her application set up. In the future, we intend to model a complete system and use our framework for its power estimation of separate running applications.

6. REFERENCES

[1] G. Banga et al. Resource containers: A new facility for resource management in server systems. In *USENIX OSDI*, 1999.

[2] T. L. Cignetti et al. Energy Estimation Tools for the Palm. In *ACM MSWIM*, 2000.

[3] M. Dong et al. Power Modeling of Graphical User Interfaces on OLED Displays. In *ACM/EDAC/IEEE DAC*, 2009.

[4] M. Dong et al. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *ACM MobiSys*, 2011.

[5] P. Dutta et al. Procrastination might lead to a longer and more useful life. In *HotNets*, 2007.

[6] P. Dutta et al. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IEEE IPSN*, 2008.

[7] J. Flinn et al. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *IEEE WMCSA*, 1999.

[8] R. Fonseca et al. Quanto: Tracking energy in networked embedded systems. In *USENIX OSDI*, 2008.

[9] K. Kim et al. Fepma: Fine-grained event-driven power meter for android smartphones based on device driver layer event monitoring. In *DATE*, 2014.

[10] D. McIntire et al. The low power energy aware processing (leap)embedded networked sensor system. In *IEEE IPSN*, 2006.

[11] D. McIntire et al. Etop: Sensor network application energy profiling on the leap2 platform. In *IEEE IPSN*, 2007.

[12] D. McIntire et al. Energy-efficient sensing with the low power, energy aware processing (leap) architecture. *ACM Transactions on Embedded Computing Systems*, 11(2), 2012.

[13] R. Mittal et al. Empowering developers to estimate app energy consumption. In *ACM MobiCom*, 2012.

[14] A. Pathak et al. Fine-grained power modeling for smartphones using system call tracing. In *ACM EuroSys*, 2011.

[15] F. Qian et al. Profiling resource usage for mobile applications: A cross-layer approach. In *ACM MobiSys*, 2011.

[16] J. A. Stankovic et al. Energy management in sensor networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370(1958), 2012.

[17] T. Stathopoulos et al. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *IEEE IPSN*, 2008.

[18] H. Zeng et al. Ecosystem: Managing energy as a first class operating system resource. In *ACM ASPLOS*, 2002.