Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices

Abhinav Pathak
Purdue University
pathaka@purdue.edu

Y. Charlie Hu
Purdue University
ychu@purdue.edu

Ming Zhang
Microsoft Research
mzh@microsoft.com

ABSTRACT

This paper argues that a new class of bugs faced by millions of smartphones, energy bugs or *ebugs*, have become increasingly prominent that already they have led to significant user frustrations. We take a first look at this emerging important technical challenge faced by the smartphones, ebugs, broadly defined as an error in the system (application, OS, hardware, firmware, external conditions or combination) that causes an *unexpected* amount of high energy consumption by the system as a whole. We first present a taxonomy of the kinds of ebugs based on mining over 39K posts (1.2M before filtering) from 4 online mobile user forum and mobile OS bug repositories. The taxonomy shows the highly diverse nature of smartphone ebugs. We then propose a roadmap towards developing a systematic diagnosing framework for debugging ebugs on smartphones.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Performance Attributes
D.2.5 [Testing and Debugging]: Diagnostics
General Terms Reliability, Measurement, Design.
Keywords Energy, Energy Bug, Ebug.

1. INTRODUCTION

Despite the incredible market penetration of smartphones and exponential growth of their apps market, their utility has been and will remain severely limited by their battery life. Consequently, compared to their desktop counterparts, smartphones face a new class of abnormal system behaviors, namely, *energy bugs*. In fact, ebugs have become so prominent that already they have led to significant user frustrations. 70% of phones returned to Motorola were related to energy problems [1].

We define an *energy bug*, or *ebug*, as an error in the system, either application, OS, hardware, firmware or external that causes an *unexpected* amount of high energy consumption by the system as a whole.[1] Unlike traditional software bugs, such errors could happen in any of the wide variety of entities in a phone (*e.g.*, hardware components, app processes, or the OS), and due to a wide variety of root causes such as programming errors, inappropriate API usage, flaws in the design of applications or the OS (device drivers), complicated interactions between hardware components of smartphones, changing external conditions (*e.g.*, remote server crash, or weakened wireless signal strength), and faulty hardware itself from daily "wear and tear" due to the portable nature of smartphones (14% of all technical support calls for Android is related to faulty hardware [2], 9% for WP7 and 8% for iPhone).

Also different from traditional software bugs, ebugs do not lead to an application crash, or OS screen of death. In most cases of ebugs, the applications and the whole system continue to run, providing their intended services normally, with the exception of consuming unexpectedly large amount of energy. This makes bug detection and root cause tracing much more difficult. The end result of an ebug is that mobile devices run out of battery sooner or much sooner than expected, requiring frequent recharging or resulting in unplanned device battery outage.

Together, these two unique characteristics of ebugs, diversity in root causes and stealth nature, suggest that they can be much harder to detect and pinpoint the root causes when compared to traditional software bugs.

This paper takes a first look at ebugs on smartphones and calls for the research community to expend research effort to tackle the associated technical challenges in treating them. We first present a taxonomy of smartphone ebugs based on mining over 39K (1.2M before filtering) posts from 4 online forums. The taxonomy illustrates the diverse nature of possible ebugs on modern smartphones in terms of their symptoms (which hardware component is physically draining energy) and causes (hardware defect, software bug, or unhan-

[1]This definition encompasses energy optimization as a special case which aims at reducing the largely expected though inefficient energy consumption of applications. However, most of the ebugs discussed in this paper result in severe battery depletion and consequently considerable user frustration.

Hotnets '11, November 14–15, 2011, Cambridge, MA, USA.

dled external events). We then propose a roadmap towards developing a systematic diagnosing framework for debugging ebugs on this unique, increasingly dominant computing platforms in the foreseeable future.

2. METHODOLOGY

To gain understanding of ebugs affecting smartphone users, we crawled Internet forums discussing smartphone energy problems and bug reports, and performed postprocessing to extract reported ebugs, and if available, root cause behavior.

Data collection. We pursued a two-pronged approach in collecting ebug data from the Internet. First, we downloaded bug reports of Android [3] and Maemo OS (Nokia) [4] reported online by users. These bug reports give more definite information about the ebugs, possible causes and fixes by developers, but have limited coverage. Second, we crawled Internet forums dedicated to mobile related discussions which provide much wider and updated coverage of the problems that users have experienced.

We scrapped 4 popular mobile Internet forums: one general forum with discussions covering all mobile devices and OSes, and three OS/company specific mobile forums. Each posting thread, or otherwise called discussion or simply *post*, is started by a user who observes a specific problem with her mobile, or wants to discuss a new feature, and is followed by emails from other users/developers. In all, we downloaded over 1.2M posts from the four online forums, with 10 emails per post on average. The posts ranged over 390 mobile devices, including smartphones, tablets, book readers, and PDAs (no laptops or PCs), with the majority related to smartphones. Over 80% of the posts were posted in the past 20 months.

Postprocessing. To extract posts related to ebugs from these forum postings, we first applied simple regex on the text of the postings to match certain keywords (such as "battery", "drain", "power", "wake" and "energy") which results in 39K posts. We next applied k-means [5] clustering to cluster these 39K posts based on the text contained in their body. We used the Lemur [6] tool for this purpose which applies standard Information Retrieval techniques like word stemming, removing stop words, building index tree, *etc.*, before clustering the documents. We took care to include the name of the devices in the stop word list to prevent clustering based on the devices being discussed.

K-means splits the 39K posts into about 1K groups. We discarded the bottom 500 groups each of which contained less than 20 posts. We then labeled each group with the theme of the postings, by manually reading the postings in the groups. We then (manually) selected those clusters where the theme encompasses the ebug symptom: unexpected high battery drain.

3. ENERGY BUGS: A TAXONOMY

We now present a taxonomy of the ebugs, shown in Figure 1, from digesting the processed trace.

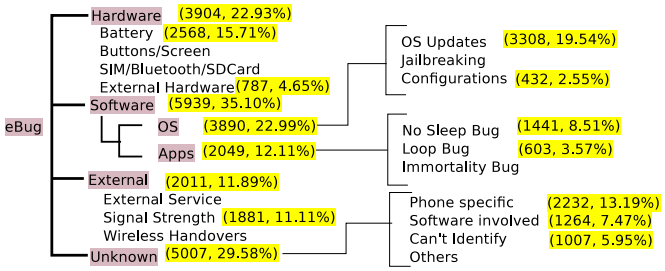

Figure 1: A taxonomy of energy bugs on smartphones. (xx, yy%) denotes that there were xx ebug posts related to the topic which amounted to yy% of all posts.

3.1 Hardware Ebugs

Battery. A large portion of users (2568, 15.71% posts) observed that severe energy depletion was due to faulty battery. A faulty battery does not hold the complete charge upon charging. Also, the charge held by the faulty battery drains internally and results in heating the battery. The posts cited several reasons for battery turning faulty, including charger damage, old battery, and water damage. Apart from the fact that the battery could get damaged resulting in holding lesser charge, it was observed that in some cases, the mobile device displayed incorrect battery statistics [7], *e.g.*, a smartphone displays 100% battery charged even though the battery is charged only 30%, triggering the user to believe a problem exists somewhere. The faulty battery problems were either solved by the phone OEM sending a new battery, or the owner buying a new one, or in some cases, using a technique called "battery calibration" [8].

Exterior Hardware Damage. Exterior damage to mobile device hardware could also result in unexplained battery drain from the device. A hardware damage can be solely responsible for the battery drain or could act as a trigger for other parts of the phone (like software) to drain battery. For example, a damage to external buttons of the phone (209, 1.23%) caused the "home" button to be oversensitive, and resulted in random unlocking of the mobile and hence turning on the backlight and CPU at various times, draining battery. Similarly, an external damage to the touchscreen of the phone rendered it too sensitive to external touches.

SIM. The SIM card of the phone can also cause battery drain (78, 0.43% posts) in multiple ways. (a) An old SIM can have several scratches or be bent inwards or outwards, increasing its internal resistance [9] or even leading to bad contacts shorting and draining battery. (b) Different SIM cards operate in different voltages (5V, 3V, 1.8V) depending on their generation and a mismatch could trigger a battery drain. (c) Usage of micro SIM in newer phones (*e.g.*, iPhone) forces users to "cut" their normal SIM, which can damage SIM, shorting its PINs, resulting in unexplained energy drain.

SDCard. An external SDCard can act as a trigger for severe battery drain. Specifically, a corrupted SDCard or portions of it could either trigger buggy apps into a looping state, where they repeatedly try to access the hardware [10], or put

them in a hanging state, while they continue to drain battery by holding up other components (no sleep bug in § 3.2.2).

External Hardware. External hardware attributed to (787, 4.65%) posts related to battery problems. Out of these, erroneous phone chargers were reported as the biggest source (698, 4.12%). Wall chargers, USB chargers, and car chargers were reported to only partially charge the phone. In some cases, battery energy depletion was observed after connecting it to the charger which considerably heated the device. External docks such as music speakers and keyboards were reported as the sources of battery drain. These external hardware usually contain their own power source or additional batteries, but were observed to drain power of the mobile device connected to them, *e.g.*, Eee-pad when connected to external keyboard was observed to lose energy rapidly.

3.2 Software Ebugs

3.2.1 OS Ebugs

OS updates, by user or forced (*e.g.*, Over-The-Air (OTA) update), represented the largest fraction (3308, 19.54% posts) of user complaints regarding ebugs[2]. IOS updates (4.0 to 4.3.3) (802, 4.74% posts) resulted in severe energy depletion in Apple mobile devices including iPhone, iPod and iPad. Android OS updates triggered battery depletion in several handsets (405, 2.39% posts) like Samsung Galaxy S [12, 13], HTC Evo [14], Nexus One [15, 16], froyo [17]. OTA updates on Android, *e.g.*, an HTC update [18], also caused battery drain. Jailbreaking was also reported as the trigger for energy depletion in (178, 1.05%) posts.

The root cause of battery drain due to an OS update could be an ebug in the updated OS itself (*e.g.*, a new OS configuration), or in the framework (*e.g.*, the Android framework), or in one of the apps that accompany the new OS (*e.g.*, widgets and bloatware pushed by phone operators). In most of the ebug postings related to OS, pinpointing the root cause of the bug is particularly hard due to the closed-source nature of some mobile OSes (*e.g.*, IOS). However, in case of other OSes, user postings indicated two categories of root causes:

OS Processes. Buggy OS processes were observed to cause battery drain. For example, the "Suspend" process in Android was observed to run in the background keeping the CPU awake and busy draining battery [14], and the "System_Server" process was observed to drain battery in the background due to high processing [16].

Configuration Changes. OS configuration changes also resulted in high battery drain. For example, a simple configuration change from the tickless to ticked kernel [19] increased power consumption; incorrect profile to SetCPU [20] for overclocking and underclocking the kernel resulted in drastic energy depletion. These configurations drained battery due to incorrect utilization of the component or bad sleeping policies.

[2]The high number of OS update postings is partly due to the fact that one of the forums used in the study is used by developers to test/distribute their own version of mobile ROMs, similar to [11], and users post their experience using the distributed OS versions.

3.2.2 Applications and Framework Ebugs

No-Sleep Bug. This is the most prominent ebug (1441, 8.51%) among application-related ebugs. As the name suggests, "no-sleep bugs" in applications erroneously do not allow at least one component of the phone to sleep, resulting in unnecessary, prolonged battery drain.

Modern smartphone OSes freeze the system after a short inactivity timeout to aggressively conserve energy. However, they provide APIs using which applications can wake up phone components irrespective of user activities. This feature is particularly useful in carrying out periodic background activities like polling and notifications, *e.g.*,`PARTIAL_WAKE_LOCK` exported by the PowerManager class in Android helps applications to ensure that the CPU is running irrespective of user activities. A *no-sleep bug* is a situation where an app acquires a wake lock for a component which wakes the component up, but does not release the lock even after the job is completed. The no-sleep bug was observed in many mobile apps, including Facebook [21], location listener [22], Google latitude [23], Google Calender, email apps [24, 25], camera app [26] (triggered by camera button), widgets [27], alarm app [28], dialer app, weather apps, SMS app, GPS based apps (including Google maps, which turn on GPS and do not turn it off even after their exit), Gallery app [29] (triggered by "no sleep" of the motion sensor), launcher apps, and audio over bluetooth [30].

A no-sleep bug is easier to detect by the user in cases where the switched-on component is easily noticeable, *e.g.*, screen (as in the dialer app and SMS app), but much harder to detect for components whose activities are not easily noticeable, *e.g.*, GPS and CPU (as in map based apps). The root cause for a no-sleep bug ranges from a simple programming mistake (*e.g.*, the programmer forgot to release the lock) to complicated reasons like race conditions prevented lock release [31], corner case conditions (*e.g.*, Android email client does not release wake locks after data connections get interrupted while syncing over 3G [32]), and *sleep conflicts* explained below.

Sleep Conflicts: Sleep conflicts arise due to aggressive sleeping policies of modern day smartphones. These policies force the CPU to aggressively sleep after an inactivity period. If a component is triggered into a high power state, and the CPU sleeps during this process, the component continues to draw high power as the 'code logic' that brings the component back to a low power state can not run until the CPU is woken up.

Sleep conflicts can occur in apps, *e.g.*, a map application turns on GPS and then the CPU sleeps due to inactivity, or in OS due to low level OEM device driver power management, *e.g.*, the WiFi NIC enters a high power state followed by a tail energy state [33, 34], the CPU sleeps, and the driver code that brings the NIC to a low power state (after a timeout) cannot run until the CPU wakes up.

Loop Bug:. A *loop bug* happens where a part of an application enters a looping state performing periodic but unnecessary tasks, draining significant battery. The periodic task could be either unnecessary network polling, processing, or use of any other component in a loop. A loop bug could be as simple as a routine calling itself by mistake [35], or could be complicated like the synchronization issue observed in the sync service in Google Calendar [36] (389, 2.30% posts) where SYNCADAPTER synced its own update with itself endlessly.

Several loop bugs appear to be triggered by not being able to handle unexpected external events, such as remote server crash, email password change, or change in remote software versions. Such conditions trigger the clients to behave erratically by repeatedly trying to connect to the remote server, or perform email authentication [37], or ping the server, draining battery on the phone.

Immortality Bug. An immortality bug is a situation where a buggy application that drains battery, upon being explicitly killed by the user, respawns (*e.g.*, by the framework [38]), enters the same buggy state, and continues to drain battery. MediaServer [39] and Google Maps [40] are some examples.

3.3 Ebugs Triggered by External Conditions

Wireless Signal Strength. Weak wireless signal strength causes NIC drivers to compensate by increasing its Tx/Rx power [34] which can significantly increase the energy drain of apps that perform network activities, *e.g.*, background processes that perform periodic polling. A total of (1881, 11.11%) posts indicated the weak wireless signal strength (over 4G, 3G, EDGE, WiFi and GPRS) as the possible root cause for unusual battery drain spikes (*e.g.*, up to 30-80% battery drain over a 10-12 hour period on several phones.)

Wireless Handovers. Another problem related to weak wireless signal is repeated network handovers which can result in severe battery depletion. In (130, 0.77%) user postings, the mobile device was observed to drain significantly higher energy due to repeated network handovers (3G to EDGE and vice versa), mostly during commutes.

While it is debatable if the above problems due to weak signal strength belong to the realm of energy optimization, we argue that the significant energy drain warrants serious "debugging" efforts.

3.4 Unknown Ebugs

Despite our best effort, the above taxonomy is potentially incomplete, as a significant portion of the postings only described the symptoms without a clear idea of the root causes. Out of the 5007 such posts, (4000, 23.63%) posts reported battery problems potentially due to specific apps: browser, juiceDefender, skype, musicplayers, taskkillers, themes, to be handset specific, due to tethering, *etc.*, while the remaining (1007, 5.95%) posts were about ebug symptoms for which users were not able to identify the cause. Such a high percentage of ebugs with unknown causes underscores the challenges faced by debugging ebugs on smartphones.

4. TOWARDS ENERGY DEBUGGING ON SMARTPHONES

Currently, systematic treatment of ebugs on smartphones does not exist, and users cope with ebugs in an ad-hoc manner. An ebug is first detected by a user through the obvious symptom: the battery drain rate suddenly becomes very high for no apparent reason. This triggers the user to hunt for the cause. Typically, the user employs task killers to kill a suspicious app, hoping to stop the energy drain. If the symptom persists, the process is repeated. A few tools exist today that help user narrow down the suspicious app, either by providing information such as the fraction of the total system energy consumed per process and by the OS (*e.g.*, the battery tool in Android), or by specialized monitoring (*e.g.*, spareparts [41] keeps tracks of CPU wake locks). The forum posts suggest many users try to use these tools in bug hunting. However, these tools fall far short in dealing with the entire diverse spectrum of ebugs (as evident from the posts), and additionally cannot pinpoint the root cause of ebugs.

4.1 Goals and Challenges

The primary goal of energy debugging on smartphones is to pinpoint the exact root cause for an unexpected high rate of energy consumption. The diversity of ebugs exemplified in the taxonomy above suggests that energy debugging on smartphones faces significant, unique challenges. First, even narrowing down the user-perceived energy drain symptom to one specific entity is nontrivial as there are so many candidate entities that can bear the symptom, including the diverse set of hardware components and external hardware (*e.g.*, chargers), OS, firmware, framework (*e.g.*, Android), apps, and external triggers such as wireless signal strength or misbehavior of remote servers the phone is interacting with. Second, even after narrowing down an energy drain to a specific entity, pinpointing the root cause (*e.g.*, which routine in the app) and developing an eventual fix is a troublesome activity due to the giant structure of the mobile programming ecosystem, which consists of app developers (*e.g.*, gmaps), framework developers (*e.g.*, Android), kernel developers (*e.g.*, linux community), firmware developers (*e.g.*, OEM), hardware manufacturers and wireless network operators. An ebug could potentially span across multiple parties in the ecosystem (*e.g.*, sleep conflicts). It is common for the blame game to spread quickly once an ebug is reported, *i.e.*, bug reports are bounced from one party to another (*e.g.*, Android OS update ebug got deflected to phone manufacturer [42]).

4.2 EDB: An Energy Debugging Framework

We make three crucial observations from the ebug taxonomy study in Section 3. (1) Although ebugs are diverse, the natural categorization of them into hardware, OS, framework, firmware, apps, and external triggers suggests a practical two-step approach: first isolating the bug to a category and then chasing it within the category. (2) For software (OS, framework, apps) bugs, there is a need for fine-grained

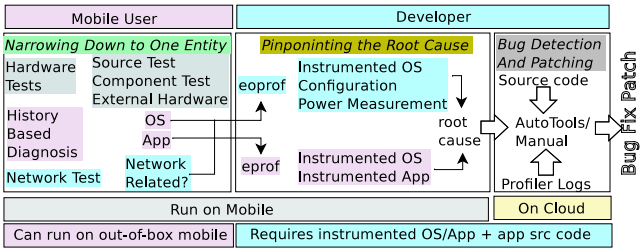

Figure 2: Components of EDB.

energy profiling and accounting, and for hardware bugs we need hardware diagnosis and monitoring. (3) Sharing information cross parties in the mobile-programming ecosystem helps to address optimization-oriented ebugs.

Following these observations, we propose a systematic framework called EDB to diagnose ebugs on smartphones. EDB has three major components, shown in Figure 2: (1) for narrowing down the ebug symptom to one entity, (2) for pinpointing the software module responsible for the ebug, and (3) automatic tools for detecting root causes. The first two components are intended to be used by mobile users to pinpoint the bug source on the mobile, while the third component (set of tools) are meant for developers to debug the apps on servers as they potentially require more processing.

4.2.1 *Narrowing Down to One Entity*

EDB employs two techniques for sequentially testing all entities on the smartphone to isolate the entity that bears the ebug symptom.

Hardware Tests. This step consists of two stages: **(a) source tests:** This stage verifies whether the source, *i.e.*, battery, is the problem by using a series of testing tool to test the battery. These would perform checks ranging from the amount of charge held by the battery after a complete charge, checking battery gauge at the OS interface and see if calibration is required, *etc.* **(b) component test:** For each of the hardware bugs in the taxonomy, including SIM, sdcard, bluetooth, we build a suite of testing tools, much like the hardware system diagnostics in the traditional BIOS for a desktop machine. The testing tools would be designed to be specific for each hardware component to administer its health, *e.g.*, a tool to check sdcard would exercise the use of the component, reading/ writing different sectors of the card while measuring the power consumed. It would then match the power consumed with the expected value.

History-based Diagnosis for Software. If none of the above hardware bug testing code catches any ebugs, the ebug must exist either in the OS (or framework), or in some application process (semantic bug or triggered by external conditions). EDB uses *history-based diagnosis* to narrow down the bug to one of these entities. The basic idea is to take a snapshot of the whole phone state information periodically, for example, on a daily basis. Upon observing an ebug symptom, *i.e.*, unexpected high energy consumption, EDB takes a snapshot of the current system and calculates the diff from the previous snapshot. A new entity, or a modified entity (entities) is (are) the prime suspect(s) of the ebug. For example, a move to a new area (which changes the wireless signal strength), an OS update, a configuration change of an app (*e.g.*, password change), installation of a new app, would be examples of isolated entities that lead to ebugs. This is similar to techniques where configurations comparisons are used to solve system problems (*e.g.*, [43]).

4.2.2 *Narrowing Down to Software Component*

For software ebugs, after narrowing down the ebug to the particular application or the OS, the next step is localize the cause to the specific software module.

Need for *eprof*. For application ebugs, we envision it is essential to develop a general-purpose call-graph *energy profiler*, *eprof*, which can be used by an app developer or mobile user to profile and consequently pinpoint the energy hog of smartphone apps. Developing such a profiler can directly leverage various online power models [33, 34, 44]. We envision the energy profiler to be call-graph-based just like *gprof* [45] as an app is typically implemented as a set of subroutines following the widely accepted modular programming design principle.

Need for *eoprof*. Finally, for OS (or framework) ebugs (see Figure 1), an *energy profiler* for the kernel, *eoprof*, much like *oprofile* [46] for performance profiling the kernel, needs to be developed. Developing *eoprof* faces several challenges. First, it relies on a power model, but an ebug in the kernel or lower level device driver could render power models such as [34, 33, 44] incorrect as a component's power consumption logic could itself be buggy. Second, current power models are designed to target modeling the power consumption of apps and hence can be too course-grained for modeling kernel activities, *e.g.*, they do not capture variations in kernel configurations. For example, a simple configuration change from the tickless to ticked kernel could result in significant energy drain [19]. To overcome these challenges, *eoprof* requires not only calibration of device drivers against ground truth power dissipation but also fine-grained power modeling that incorporates kernel/framework configs.

4.2.3 *Automatic Tools for Root Cause Detection*

Like *gprof*, the *eprof* and *eoprof* tools discussed above are semi-automatic debugging tools; they finally rely on developers' knowledge to uncover the cause that led to *why* a particular routine consumes high energy. Developing automatic energy debugging tools is an important topic but has been barely explored. We discuss some initial ideas towards treating the software bugs listed in § 3.2. For "no-sleep bugs", the root cause is the mismatch between acquiring and releasing of wake locks, two actions that can be programmed in different subroutines of an app or in the OS framework. This resembles concurrency bugs in debugging concurrent programs (*e.g.*, [47]), and both techniques based on static analysis (*e.g.*, [48]) and dynamic analysis (*e.g.*, [49]) can be developed to automatically discover such bugs. For "loop bugs", the root causes are often unexpected external events not properly handled by the application. One approach to de-

tecting such bugs is to apply model checking (*e.g.*, [50, 51]) or symbolic execution (*e.g.*, [52]) previously developed for or applied to detecting errors in distributed systems to automatically check for the program behavior under possible external events.

4.2.4 Cross-Layer Solutions

Addressing optimization-flavored energy problems such as the wireless signal related ones in §3.3 requires a concerted approach across multiple parties in the mobile-programming ecosystem. For example, under weak signal strength, there exists a tradeoff between consuming more energy versus giving up network connectivity to save energy, and the correct tradeoff is likely to depend in the importance of the apps and user preference. This calls for sharing information between the firmware, the OS, the app, and possibly user configurations. Other examples include sleep conflicts and the immortality bug.

5. RELATED WORK & CONCLUSION

Energy debugging on smartphones is related to the body of work on software debugging on desktop and servers and can draw ideas from this area either on failure diagnosis techniques such as source code tracing, system logs, and execution replay [53, 54], and on performance debugging techniques such as history-based analysis [55], resource accounting [56], and blackbox debugging [57]. Recently, Mobibug [58] was proposed as a diagnosis framework to debug mobile application crashes.

This paper takes a first look at ebugs, a new class of bugs faced by millions of smartphones, that have already led to significant user frustrations. We presented a taxonomy of the kinds of ebugs based on mining over 39K posts (1.2M before filtering) from 4 online mobile user forums and mobile OS bug repositories. The taxonomy shows the highly diverse nature of smartphone ebugs. We further proposed a roadmap towards developing a systematic diagnosing framework for treating these ebugs. We argue energy debugging is an important and challenging research area that warrants significant timely efforts from the research community.

Acknowledgment. Abhinav Pathak was supported in part by the 2011 Intel PhD Fellowship.

6. REFERENCES

[1] "The bulk of returned motorola phones are the result of applications that cause performance/energy problems." URL: http://tinyurl.com/44hns9p
[2] "Android phones more prone to hardware problems." URL: http://www.pcmag.com/article2/0,2817,2387493,00.asp
[3] "Android - an open handset alliance project." URL: http://code.google.com/p/android/issues/list
[4] "Maemo community." URL: http://maemo.org/intro/
[5] "k-means clustering." URL: http://en.wikipedia.org/wiki/K-means_clustering
[6] "The lemur project." URL: http://www.lemurproject.org/
[7] "Incorrect battery reading on droid (2.0)." URL: _cgc_5259
[8] "Apple portables: Calibrating your computer's battery for best performance." URL: http://support.apple.com/kb/HT1490
[9] "Meamo.org: Make your battery last longer." URL: http://wiki.maemo.org/Make_your_battery_last_longer
[10] "Sd card corruption." URL: _cgc_2500
[11] "Cyanogenmod: Android community rom based on froyo." URL: http://www.cyanogenmod.com/
[12] "Android os 2.3.3 battery drain." URL: _cgc_16721
[13] "After upgrade galaxy s to gingerbread 2.3.3, suffering a huge battery drain." URL: _cgc_16562
[14] ""suspend" process runs continually in background at 40% cpu on htc evo 4g." URL: _cgc_11126
[15] "Heavy battery drain after gingerbread upgrade on nexus one." URL: _cgc_15057
[16] "Cpu is busy and battery drains." URL: _cgc_9733
[17] "The process "/ init" uses between 70% to 98% cpu - htc legend froyo." URL: _cgc_13130
[18] "User posting: 2.2 update - htc confirms major problems." URL: http://androidforums.com/htc-droid-incredible/167864-2-2-update-htc-confirms-major-problems.html
[19] S. Siddha, V. Pallipadi, and A. Ven, "Getting maximum mileage out of tickless," in *Proc. Linux Symposium*, 2007.
[20] "Setcpu for root users." URL: http://www.setcpu.com/
[21] "Facebook 1.3 not releasing partial wake lock." URL: http://geekfor.me/news/facebook-1-3-wakelock/
[22] "Using a locationlistener is generally unsafe for leaving a permanent partial_wake_lock." URL: _cgc_4333
[23] "Latitude prevents nexus s from sleeping." URL: _cgc_17356
[24] "Email application partial wake lock." URL: _cgc_9307
[25] "E-mail app has a bug which causes a partial wake lock to be held until manually interrupted." URL: _cgc_6811
[26] "When locked, the camera button wakes up the system causing battery to drain fast on samsung galaxy." URL: _cgc_4293
[27] "'power control' widget brightness toggle leaves g1 keyboard backlight on." URL: _cgc_4211
[28] "After dismissing alarm clock, screen does not timeout." URL: _cgc_6917
[29] "Do you suffer from gallery sensor battery drainage?" URL: http://forum.xda-developers.com/showthread.php?t=838949
[30] "Huge battery drain because bt-headset sound is not stopped." URL: https://bugs.maemo.org/show_bug.cgi?id=9640
[31] "Locationmanagerservice: Fix race when removing locationlistener." URL: https://review.source.android.com/#/c/12110/
[32] "Email 2.3 app keeps awake when no data connection is available." URL: http://www.google.com/support/forum/p/Google+Mobile/thread?tid=53bfe134321358e8
[33] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proc. of CODES+ISSS*, 2010.
[34] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system-call tracing," in *Proc. of EuroSys*, 2011.
[35] "Fix bugs in audiostreaminalsa::close() and audiostreamoutalsa::close()." URL: https://review.source.android.com/#/c/13467/
[36] "Google calendar sync problem, continuously tries to sync, drains battery quickly." URL: _cgc_6107
[37] "Repeated email sync failure eats cpu and battery." URL: _cgc_5424
[38] "Android service." URL: http://developer.android.com/reference/android/app/Service.html
[39] ""mediaserver" consuming 100% cpu time after failing to play streaming videos." URL: _cgc_6765
[40] "Maps continually running causing battery drain." URL: _cgc_10790
[41] "Spare parts." URL: https://market.android.com/details?id=com.androidapps.spare_parts
[42] "Android os battery use excessive - short battery life." URL: _cgc_14684
[43] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. Wang, C. Yuan, and Z. Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," in *LISA*, 2003.
[44] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Mobisys*, 2011.
[45] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *Proc. of ACM PLDI*, 1982.
[46] "Oprofile." URL: http://oprofile.sourceforge.net/news/
[47] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes — a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.
[48] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, 2010.
[49] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *SOSP*, 2007.
[50] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM Trans. Comput. Syst.*, 2006.
[51] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "Crystalball: Predicting and preventing inconsistencies in deployed distributed systems," in *Proceedings of NSDI*, 2009.
[52] O. Crameri, R. Bianchini, and W. Zwaenepoel, "Striking a new balance between program instrumentation and debugging time," in *Proceedings of Eurosys*, 2011.
[53] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," in *SOSP*, 2007.
[54] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," *ASPLOS*, 2010.
[55] P. Bodik, M. Goldszmidt, A. Fox, D. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *Eurosys*, 2010.
[56] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI*, 2004.
[57] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *SOSP*, 2003.
[58] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl, "There's an app for that, but it doesn.t work. diagnosing mobile applications in the wild," in *Hotnets*, 2010.

***macro** _cgc_= "http://code.google.com/p/android/issues/detail?id="