Reducing Energy Consumption of Alarm-induced Wake-ups on Android Smartphones

Sewook Park, Dongwon Kim, Hojung Cha
Department of Computer Science
Yonsei University
Seoul, Korea
{swpark,dwkim,hjcha}@cs.yonsei.ac.kr

ABSTRACT

Alarms are often used to set smartphones to perform tasks at scheduled times. Many applications use alarm functionality, and devices consequently experience frequent wake-ups and waste energy. In this paper, we analyze alarm-induced wake-ups in the Android platforms in terms of energy consumption. We propose a "Time Critical Alarm", in which alarms necessarily accompany wake-ups. We then propose, AlarmScope, a scheme to reduce non-critical alarms and thus minimize energy waste. Our evaluation of widely-used applications on Android smartphones shows that the proposed scheme would save between 2.6% and 12.5% of energy use.

Categories and Subject Descriptors

D.2.11 **[Software Engineering]**: Software Architectures; D.2.8 **[Software Engineering]**: Metrics—Performance Measures

General Terms

Design, Experimentation, Measurement

Keywords

Smartphones, Mobile, Energy, Alarm, Wake-up

1. INTRODUCTION

As the number of applications installed on smartphones increases, energy waste can become significant due to the inefficient operations of these applications. Recently, active work has been conducted to deal with this issue. Pathak et al. [1] analyzed various types of energy bugs arising in smartphone applications, and Ma et al. [2], using eDoctor, examined the causes of the abnormal draining of smartphone batteries. Zhang et al. [3] conducted a comparative analysis of energy bugs on various smartphone platforms. Oliner et al. [4] identified the types of energy anomalies with massive amounts of user data via Carat and proposed a method with which to address energy anomalies.

Among the approaches to reducing energy inefficiency in mobile devices, sleep functionality has, in particular, been treated as an important issue when dealing with energy waste. Sleep is generally defined as a state in which the smartphone OS freezes the system to save energy. Many works have addressed the energy bug, which refers to a state in which the system refuses to enter sleep mode or energy is constantly drained even in sleep mode. Prior works [5, 6] have proposed ways of locating no-sleep bugs at the source-code level during compile time based on static code analysis. Kim et al. [7] dynamically analyzed no-sleep bugs that were undetected during compile time, with no modification to the system. Liu et al. [8] addressed the issue of energy inefficiency caused by the abnormal behavior of sensors while in sleep mode. A method [9] was proposed for detecting a condition in which the smartphone is unable to enter the suspend mode at the device-driver level. Studies of the no-sleep bug analyzed the causes that prevent devices from entering into sleep mode, suggesting methods of detecting conditions that give rise to no-sleep bugs. In summary, the works published so far have focused on the phenomenon in which the system does not enter *sleep from wake-up* mode. However, no previous work has addressed the issue of the inefficiency of the transition from *sleep to wake-up* mode.

A smartphone generally wakes from sleep upon user input, such as pressing the power button. The smartphone may also wake up via network requests, and more explicitly, applications activate alarms via Alarm Manager. Here, unnecessary wake-ups induced by alarms may occur when an alarm type has been incorrectly assigned or alarms are used excessively. When many applications are installed on a device, wake-ups can be called frequently, resulting in an increased chance of creating unnecessary alarms. In order to manage energy efficiently on smartphones, a policy is therefore needed to address the inadequate use of alarms caused by developers.

In this paper, we propose a method, named AlarmScope, that prevents nonessential wake-ups. We first analyze the internal behavior of each alarm, as well as the applications, when a wake-up occurs via an alarm. We then investigate the energy cost for wake-ups and decide whether this cost could be reduced. We define a term, Time Critical Alarm (TCA), in order to identify and exclude alarms that are not critical to the operation of applications. AlarmScope converts non-deferrable alarm judged as non-critical to deferrable. With its implementation on the Android smartphone, we validate the effectiveness of the proposed method and demonstrate that our method is practical in terms of saving energy.

2. BACKGROUND AND MOTIVATION

We briefly explain the internal mechanism of an alarm on the Android platform. We then discuss the issues caused by the inefficient use of alarms triggered by applications.

2.1 Android AlarmManager

An application is allowed to schedule a task at a desired time through Alarm. In order for the device to exit sleep mode, AlarmManager must be used. When an application registers an alarm on the Android platform, the type of alarm and the alarm

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
HotMobile '15, February 12 - 13 2015, Santa Fe, NM, USA
Copyright 2015 ACM 978-1-4503-3391-7/15/02 $15.00
http://dx.doi.org/10.1145/2699343.2699346

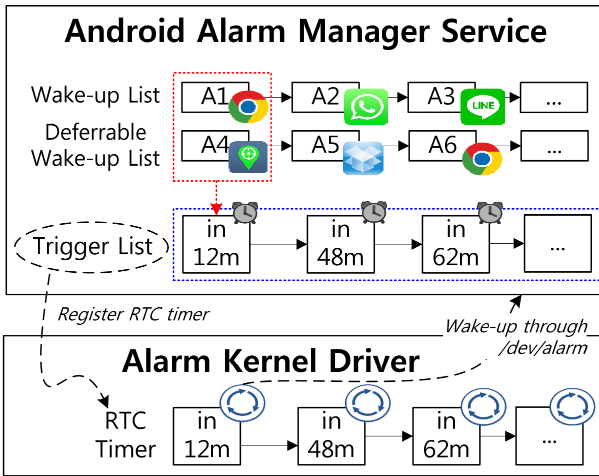


Figure 1. Android AlarmManager architecture

time are registered. Two types of alarms exist: non-deferrable alarms, which should run at the specified time, and deferrable alarms. When an alarm is registered, as shown in Figure 1, AlarmManager Service composes a list of alarms to be executed, and it registers the time points on the hardware RTC, so that the device can wake up from sleep via the kernel timer in the alarm driver. At the set time, the alarm driver writes the relevant data into /dev/alarm; the alarm library detects this and relays the data to the alarm handler in AlarmManager Service.

This mechanism does not, however, consider the energy cost of using alarms. KitKat (Android 4.4) has newly introduced the concept of Alarm Batch to improve the energy efficiency of the system. When multiple alarms are set within a close time range, they are processed in a batch at their set times.[1] This removes the need for unnecessarily frequent wake-ups in the native Android framework, but we assume it still has energy inefficiency.

2.2 Preliminary Experiment

We conducted an experiment to measure the energy consumption caused by alarms in commercial applications. The smartphone device selected was the Nexus 4 running Android 4.3 JellyBean and 4.4.2 KitKat, with 15 commercial applications installed. Among the free applications available in Google Play, we chose the ones that make frequent use of wake-up alarms. These were selected from three categories: Communication (Line, Kakao Talk, MyPeople, WeChat, KeeChat), Social (Naver Band, Naver Blog, Naver Café, Tumblr, Between), and Tools (Dodol Phone, Memoy Booster, Battery Doctor, Clean Master, Osmino WiFi). The experiment was conducted while the screen was turned off and the applications were running in the background.

Figure 2 shows the power trace before and after the smartphone enters the wake-up mode from sleep. The device used 26 (±7) mW in sleep and 209 (±100) mW during wake-up. During a wake-up induced by an alarm, the smartphone consumes 8 times more energy than in sleep mode, and frequent wake-ups significantly worsen battery efficiency.

[1] A similar method called 'timer coalescing' has been used to reduce energy consumption in a general-purpose OS. In the Linux kernel, the concept of a deferrable timer is used to keep the CPU idle as much as possible. In Microsoft Windows 7 and Mac OS X Mavericks, timers are put in groups to save energy on portable computers [10].

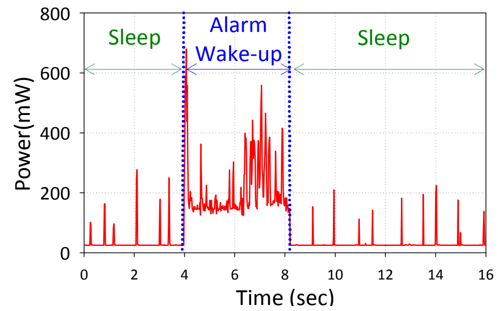


Figure 2. Power trace when an alarm wake-up occurs

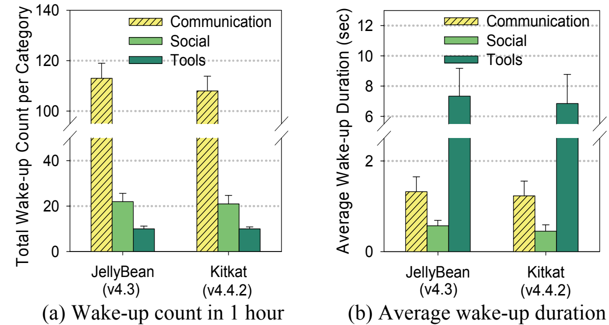


Figure 3. Alarm wake-ups for 15 Apps in three categories

In our experiment, we counted the instances of wake-ups for an hour and measured their durations. The alarm option in dumpsys, provided in the Android platform, was used for this measurement. Energy consumption was measured with the Monsoon Power Monitor.

The number of wake-ups caused by alarms was 145 (±9) on JellyBean and 139 (±10) on KitKat. Figure 3(a) shows the number of alarm-induced wake-ups for each application category. Communication applications produced the highest number of alarms, while applications in the Social and Tools categories invoked alarms relatively less frequently. Figure 3(b) illustrates the fact that the average wake-up duration was longest for applications in the Tools category. The energy consumed on Nexus 4 with JellyBean was 196,000 (±15,000) mJ before the 15 applications were installed, and 266,400 (±16,000) mJ after installation, which is greater by 35.9%.

Our experiment backs up the claim that in order to reduce energy waste, the system should play an active role in managing alarms, instead of giving full permission to developers to use alarms.

3. ALARMSCOPE

To efficiently reduce the energy cost caused by alarm-induced wake-ups, each non-deferrable alarm must be examined to determine whether it can be made deferrable. Unfortunately, it is not an easy task to determine whether an alarm can be converted from non-deferrable to deferrable. In order to decide if the type of a given alarm was intended by the developer or assigned by mistake, a well-designed strategy is required. We have taken an intuitive approach to devise such a method.

3.1 Time Critical Alarm

We define a Time Critical Alarm (TCA) as a type of an alarm that must be activated exactly at the designated time. The following factors are considered when determining whether an alarm is a TCA: whether UI updates are involved and whether data transmission takes place after the alarm is invoked. Also, the

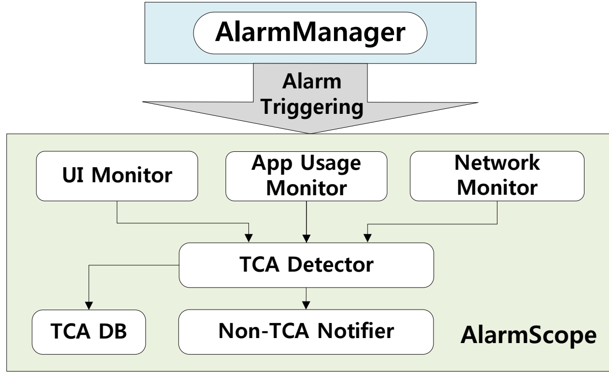


Figure 4. AlarmScope architecture

frequency of application use is given extra weight in the decision. Conditions of this kind are time-sensitive and may affect the exact operation of applications. We assume that the remainder of the alarms, which are not TCAs, are not problematic, even though there is a slight delay in their execution. The criteria for determining whether an alarm is a TCA — especially for UI updates and data transmissions — are based upon our analysis of the Android source code. This is explained in detail in Section 4.

3.2 AlarmScope Architecture

To categorize an alarm as a TCA, we developed an alarm monitoring system and named it AlarmScope. Figure 4 illustrates how the system operates on the Android platform.

AlarmScope consists of the following components: UI Monitor, App Usage Monitor, Network Monitor, TCA Detector, TCA DB, and Non-TCA Notifier. AlarmScope is activated when an alarm set in AlarmManager is triggered and relayed to an application. UI Monitor determines whether the screen is updated in any application, and App Usage Monitor keeps track of the foreground usage time for each application. Network Monitor determines whether data are transmitted from applications via the network. Based on application behavior data collected with UI Monitor, App Usage Monitor, and Network Monitor, TCA Detector decides whether a given alarm is a TCA, in which case the data are stored in TCA DB. If the alarm is a non-TCA, the data are sent to AlarmManager via Notifier.

3.3 TCA Decision Procedure

Figure 5 illustrates the procedure that determines whether an alarm is a TCA.

(1) When an application begins to run after an alarm expires, the UI update of the application is inspected via Activity. If updated, the alarm is determined to be a TCA because an immediate response to user action is required.

(2) If there is no UI update, the application usage time is considered in the TCA decision. The application usage rate is calculated using the following equation for each application. Note that the Launcher application is not considered here.

$$\text{App_usage_rate} = \frac{\text{Usage time length of App}}{\text{Sum of usage time length of each app}}$$

(3) TR-TCA_usage_rate is a threshold of critical application usage rate. If the App_usage_rate is greater than the TR-TCA_usage_rate, the alarm is a TCA. If the usage rate is less than the TR-TCA_usage_rate and also less than the TR-NonTCA_usage_rate, the alarm is a non-TCA.

(4) If the value of App_usage_rate lies between the TR-TCA_usage_rate and the TR-NonTCA_usage_rate, whether data transmission in involved is determined. The TR-NonTCA_usage_rate is used for applications that are rarely run in the foreground. An application with an App_usage_rate value lower than this threshold typically has significantly less usage time compared to other applications. The alarms in applications with less TR-NonTCA_usage_rate are negligible to the user experience. We set the value for the TR-NonTCA_usage_rate at 0.1% after a series of extensive experiments. Eight smartphone users were polled regarding application usage, and about 20% of their applications fell within this category. On the other hand, TR-TCA_usage_rate is used for frequently-used applications, and when to apply this threshold is determined by Algorithm 1.

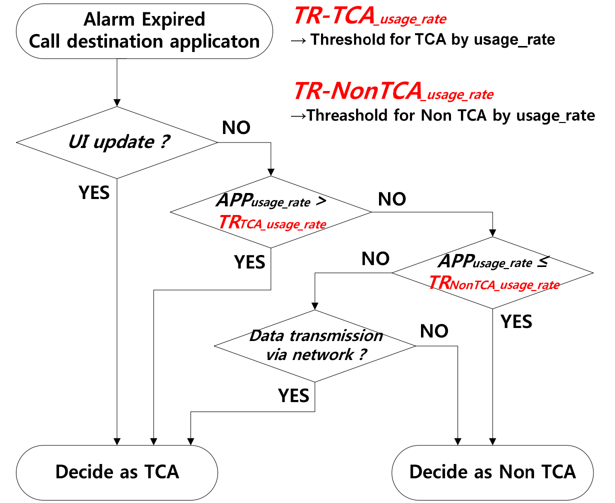


Figure 5. TCA Decision Flow

Algorithm 1 Decide TR-TCA_usage_rate

Input : Application List sorted by usage_rate in ascending order
Output : TR-TCA_usage_rate
Initial_value ← average(App $_{\text{usage_rate}}$)
App$_i$ ← First App whose usage_rate is larger than Initial_value
while App remains in the list **then**
  if (App$_{i+1\ \text{usage_rate}}$ - App$_{i\ \text{usage_rate}}$) > $\sum_{i=0}^{n}$ App$_{i\ \text{usage_rate}}$
    return App$_{i\ \text{usage_rate}}$
  else
    increase i
  endif
endwhile
return App$_{i\ \text{usage_rate}}$

Algorithm 1 decides what rate to apply to the TR-TCA_usage_rate for a given application based on how frequently the application is used. Applications sorted by usage rate are considered in ascending order during the TCA decision. If the difference between the Appi_usage_rate and the Appi+1_usage_rate is greater than the sum of usage rates

considered up to the current application, the TR-TCA_usage_rate is used for the next application.

The logic is based on the idea that the usage time differs greatly between those applications that are frequently used and those that are not. The actual frequency of application usage is reflected in the TCA decision, so the algorithm becomes effective.

4. IMPLEMENTATION ON ANDROID

Note that the newly introduced Alarm Batch functionality in KitKat does not guarantee the exact start time of the alarm. If an alarm must be executed at a precise point in time, the setExact() API must be called. We looked at the modules that use alarms in the open source code of Android and sought out where the setExact() API was called; from this information, we were able to infer which alarms were time critical.

Only five modules call setExact(): SyncManager, TwilightService, WifiStateMachine, Calendar and DeskClock. All of them perform time-critical tasks; however, the cases directly affected by applications occur when data are transmitted from an application, such as data sync, and when the screen is updated, for example, in Calendar or DeskClock. Among the many alarms used in Android open source applications, setExact() is not called except in the above five cases. We thus conclude that postponing alarms would not necessarily produce problems. Our implementation was based on this definition of TCA.

4.1 Platform Modification

When activating an alarm in AlarmManager, the relevant data registered in the alarm are verified and relayed to their destination application. An alarm contains this data internally in the PendingIntent class. We made modifications so that our defined Intent is broadcast when the send() method in the AlarmManagerService is called to relay data to an application. The AlarmScope Service receives this Intent and processes it.

The three criteria for a TCA decision are (1) UI update, (2) application usage, and (3) data transmission. We determined the criteria as follows. When the UI is updated, the finishDrawing() method in the ViewRootImpl class is called, which sends data to the WindowManager to refresh the screen. We used the UsageStatService for Application usage. The UsageStatService class keeps track of the time spent in foreground applications, and this record is stored when the application is paused. When data are transmitted, a few methods are called in classes related to socket implementation. At these points, we made code modifications which broadcast Intents to signal these conditions, which are defined separately.

If the alarm is determined to be a non-TCA event, the AlarmScope Service notifies the AlarmManager about non-TCA alarms with an Intent. When non-TCA alarm data are sent from the AlarmScope Service, the AlarmManager stores the data in memory. When a non-deferrable alarm that belongs to the non-TCA set is registered, the alarm is converted into the deferrable type. In detail, AlarmScope considers the alarms called by all APIs in AlarmManager (i.e., set(), setRepeat(), setExact(), etc.). We converted non-deferrable alarms with RTC_WAKEUP and ELAPSED_REALTIME_WAKEUP into RTC and ELAPSED_REALTIME, respectively.

4.2 AlarmScope Service

AlarmScope Service manages alarm data, which are sent from AlarmManager in the form of a list. After receiving additional Intent, AlarmScope checks whether the UI is updated or data is transmitted during a specific time. When the UI is not updated or data are not transmitted until timeout occurs, the alarm is removed from the list and is determined to be non-TCA. In this case, the application data are stored in DB and relayed to AlarmManager via Notifier. The content in AlarmScope DB is sent to AlarmManager upon rebooting so that the data are not lost. Labeling an alarm TCA is performed for each Action. In order to support such calls, each TCA is treated on a per-Action basis and is also stored as an Action.

4.3 Implementation using Xposed

As explained above, modifications must be made on the platform level in order to implement AlarmScope. Additionally, an AlarmScope service application is needed to process the relevant data during runtime. Altering the platform is time-consuming because it requires changes to the open source and generating images. To apply the modifications easily, we used Xposed [11] for implementation. On the Android platform, Xposed hooks into method calls and makes it possible to control pre- and post-method calls. Through the Xposed module, we made implementations of Intent transmission in the PendingIntent class, ViewRootImpl class, UsageStatsService class, and other socket-implementation-related classes.

Xposed runs only on rooted devices, and it is not possible to install AlarmScope on non-rooted devices. However, Xposed allows us to apply the functionality directly, without modifying the system platform, and this enables the smooth implementation of our prototype. Xposed resides on the Android Dalvik layer; thus, there is a challenge in that data transmission via socket using the native library is not detectable. Fortunately, however, we could verify that data were transmitted via the Java socket class during our preliminary experiment.

5. EVALUATION

We evaluated AlarmScope to validate its implementation and effectiveness for saving energy. We tested our implementation on Nexus 4 smartphones running Android 4.3 Jelly Bean and 4.4.2 KitKat.

5.1 Validation

We first tested whether an alarm worked correctly with AlarmScope. The test application generates non-deferrable alarms. It also periodically generates non-TCA alarms, in which neither UI updates nor data transmission via the network takes place. Then, wake-ups were traced for evaluation. In our experiment, we configured the system so that only user-installed applications were affected by AlarmScope, not the default applications shipped with the Android system.

Figure 6(a) shows the example of non-deferrable alarms delayed by AlarmScope with an interval of 30 seconds. Figure 6(b) shows the alarms and wake-ups after AlarmScope has been applied. It shows that non-TCA alarms are determined to be deferrable and hence are executed when wake-up takes place at a later point. AlarmScope gains energy benefit with the aggregated alarms due to the delay (e.g., 136 seconds and 170 seconds in Figure 6(b)). An examination of the validation data reveals that the average delay is nine seconds. The TCA-type alarms after which critical operations take place were determined to be non-deferrable and were processed at the set time. Note that the alarms generated by the Android platform and the wake-ups by network push are not delayed, and the wake-up time of alarms are not exactly the same in Android. The number of aggregated alarms is not restricted, and multiple alarms are cascaded till the next wake-up. However,

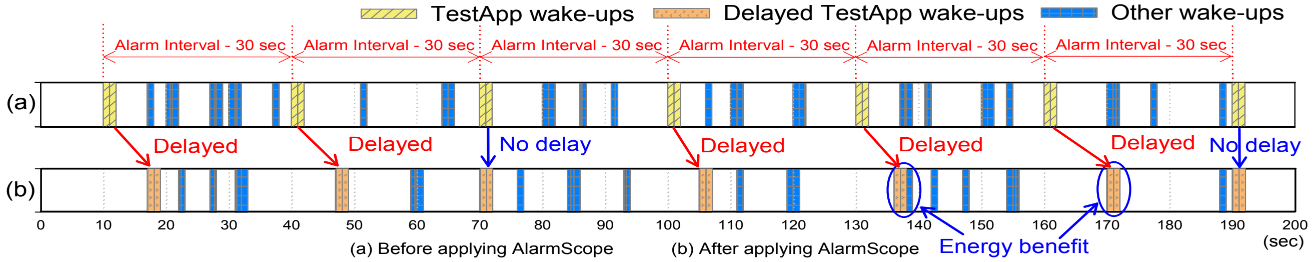
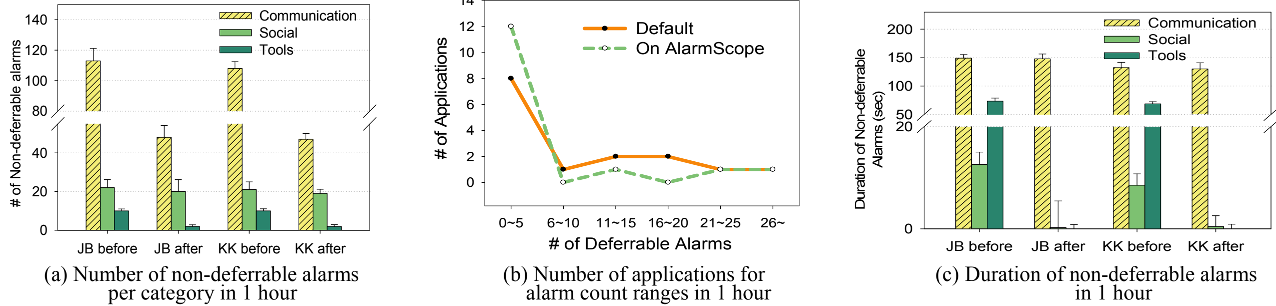


Figure 6. Trace results for Alarm-induced wake-ups



(a) Number of non-deferrable alarms per category in 1 hour

(b) Number of applications for alarm count ranges in 1 hour

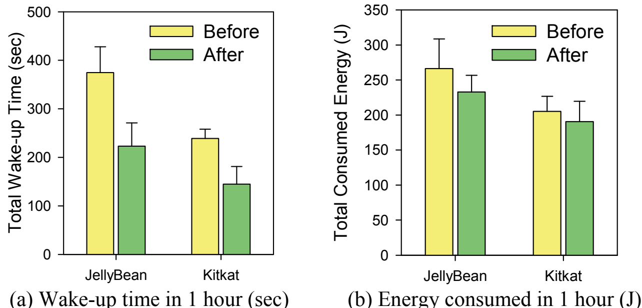
(c) Duration of non-deferrable alarms in 1 hour

Figure 7. Effectiveness of AlarmScope

the maximum delay of a non-TCA alarm is about 60 seconds because the Android AlarmManager generates wake-up every minute.

5.2 Effectiveness

We investigated the effectiveness of AlarmScope using Xposed, both before and after applying AlarmScope. As in the preliminary experiment, 15 identical applications were installed, and wake-ups were counted for each category. Measurements of the entire wake-up duration of the device, as well as power consumption, were taken. As shown in Figure 7(a), the number of non-deferrable alarms decreased from 145(±9) to 67(±5) on JellyBean, and it dropped from 139(±10) down to 64(±5) on KitKat. Overall, the number of non-deferrable alarms was reduced by more than 50%. Figure 7(b) shows the number of applications for various alarm count ranges after AlarmScope was applied. The overall non-deferrable alarm counts were reduced, and most applications induced fewer than five alarm wake-ups in one hour. In Figure 7(c), the wake-up duration of non-deferrable alarm for each category is charted. The applications in the Communication category did not show any significant decrease, but those in the Social and Tools categories had meaningful reductions in their wake-up durations. This is due to the fact that the average wake-up duration is very short for non-TCA alarms generated by an application in the Communication category; thus, the reduction in this category is very small. On the other hand, the applications in the Social and Tools categories had much reduced wake-up durations because their non-TCA alarm-induced wake-up durations are relatively longer, despite a lesser degree of reduction in alarm counts.



(a) Wake-up time in 1 hour (sec)

(b) Energy consumed in 1 hour (J)

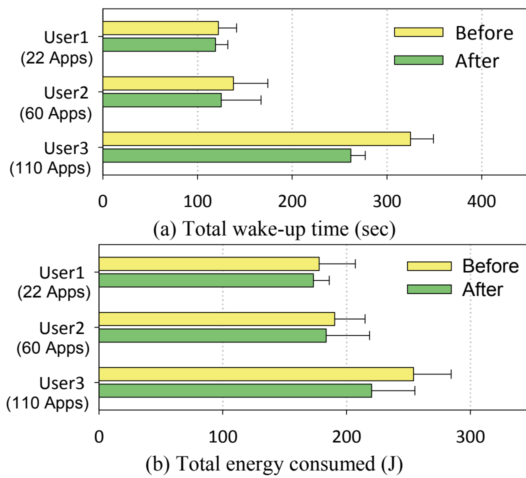
Figure 8. Wake-up time and consumed energy in 1 hour

Figure 8(a) shows the sum of the wake-up durations caused by alarms in one hour. For our experiment, we recorded the timestamps in log files when the device woke up from sleep mode. We could observe that the total wake-up time is reduced by approximately 25%. The device does not wake up when it is time to process deferrable-type alarms. Rather, it waits until the system wakes up because a non-deferrable alarm or some other triggering event induces a wake-up, at which point the deferred alarm is processed. As shown in Figure 8(b), the energy consumed in an hour was reduced by 12.5% in JellyBean and by 7.2% in KitKat. The energy reduction comes from the increased number of grouped alarms in AlarmScope. The number of grouped alarms with AlarmScope is 1.78(±0.17) in JellyBean and 2.21(±0.20) in KitKat, while the default Android shows 1.53(±0.12) and 2.03(±0.14) grouped alarms in JellyBean and KitKat, respectively.

5.3 User Experiment

We conducted an additional experiment to measure how effective AlarmScope is in practical use. Three participants were chosen in our laboratory based on the number of applications installed. We reproduced the user environments on our test smartphone and installed AlarmScope on it to measure wake-up duration and energy consumption. KitKat was installed on the test device, and we made measurements for an hour. The numbers of applications installed on the phones were 23, 60, and 110, respectively. The numbers of applications that used wake-up alarm were three, seven, and 10 for each.

The measurements derived from the test are shown in Figure 9. The reductions in total wake-up time were 2.5%, 9.4%, and 19.3% for User1, User2, and User3, respectively. The reductions in energy consumption were 2.6%, 3.5%, and 13.3%, respectively. The reason for the significant savings in User3's device is that the amount of energy consumption at wake-up is quite large and the alarms with longer average wake-up durations were converted into the deferrable type, resulting in a large savings. We verified that AlarmScope tends to function more efficiently when the number of installed applications is large.



(a) Total wake-up time (sec)

(b) Total energy consumed (J)

Figure 9. User experiment

5.4 Effects of Delayed Alarms

To investigate the side-effects of AlarmScope, we analyzed the alarm-related operations in 15 applications used in the preliminary experiments. We decompiled the apk file by dex2jar [12] and JD-GUI [13] to analyze the applications in code level. Table 1 summarizes the alarm-related operations according to the application type. In communication app types (e.g., Kakao Talk), the application used wake-up alarms to communicate with a server, to run specific services, or to register push services of applications. In social app types (e.g., Naver Band, Between, Tumblr), the alarm is used for checking the application version or deleting cache information. In tool app types (e.g., Dodol phone, Memory Booster, or Osmino WiFi), the wake-up alarms update the data, clean-up memory, or download data from server.

Based on the code analysis, we found that most alarms in deployed applications are not time-critical. The alarm-related operations are possibly categorized into (1) stand-alone operation (without server), and (2) communication operation with a server. In the case of stand-alone operation, the side-effect of a deferred alarm is trivial. For example, MemoryBooster cleans up cache information every two hours. Delaying such operations by several seconds has negligible effect on the user experience. In the case of communication operations with a server, the side-effect is also negligible, since the maximum delay time of an alarm is about 60 seconds as described in Section 5.1. In fact, no applications used the setExact() function provided by the Android AlarmManager in the latest version. We further investigated 30 most-popular applications in Google Play, and found that only one application used the setExact() function. In summary, we expect that delaying an alarm would not be critical to the user experience in a practical sense; hence its side-effects are minimal.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed AlarmScope, with which we detected unnecessary alarm-induced wake-ups on the Android smartphone and converted the alarms into deferrable types to reduce energy waste. According to [14], the average number of applications installed by a given user on a smartphone is about 26 (year 2013), and this number is increasing. Our user study suggests that applying the proposed system would reduce the energy consumption and extend the average usage time of a mobile device between charging periods.

Our method is a viable solution to manage wake-ups caused by the inefficient use of alarms, but we plan to consolidate our work further. One feasible approach is tracing the actions that follow the triggered alarms thoroughly. For example, if the final actions are checked by taint-tracking alarms, as in TaintDroid [15], the TCA decision could be further refined. The issue of false positives and false negatives will certainly be addressed in our future work. False positives, which call-out a non-time critical alarm as a TCA, need to be minimized although these would not induce system malfunctions. False negatives, which are time critical alarms that are labeled as non-TCAs, may cause system malfunctions, but AlarmScope can ignore the alarms called by time-critical function, such as the setExact() function in Android, for preventing false negative cases.

Table 1. Alarm-related operations

Category	Alarm-related operations	Example applications
Communication	Communication with server Registration of push service	Kakao Talk
Social	Check version upgrade Delete cache	Naver Blog Memory Booster Tumblr
Tools	Update data Download data from server	Dodol phone Osmino WiFi

7. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea grant funded by the Korean government, Ministry of Education, Science and Technology (No. 2014-R1A2A1A1-1049979).

8. REFERENCES

[1] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proc.of HotNets '11*, ACM Press (2011), p. 5.

[2] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proc. of NSDI '13*, USENIX (2013), pp. 57-70.

[3] Zhang, J., Musa, A., Le, W., 2013, A Comparison of Energy Bugs for Smartphone Platforms, In *Proc. of ICSE* '13, IEEE (2013), pp. 25-30.

[4] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica. Collaborative energy debugging for mobile devices. In *Proc. of HotDep* '12, USENIX (2012), p 6-6.

[5] A. Pathak, A. Jindal, Y. Charlie Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of MobiSys '12*, ACM Press (2012), pp. 267-280.

[6] P. Vekris, R. Jhala, S. Lerner, Y. Agarwal, Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs. In *Proc. of HotPower '12,* USENIX (2012), p. 3-3.

[7] K. Kim and H. Cha. WakeScope: Runtime WakeLock anomaly management scheme for Android platform. In *Proc. of EMSOFT '13*, IEEE (2013), p. 27.

[8] Y. Liu, C. Xu, and S. Cheung. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *Proc. of PerCom '13*, IEEE(2013), pp. 2–10.

[9] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *Proc. of EuroSys '13*, ACM Press (2013), pp. 253-266.

[10] Timer coalescing, http://en.wikipedia.org/wiki/Timer_coalescing

[11] Xposed, http://repo.xposed.info/

[12] dex2jar, http://code.google.com/p/dex2jar/

[13] JD-GUI, http://jd.benow.ca/

[14] The Average Smartphone User Has Installed 26 Apps, http://www.statista.com/chart/1435/top-10-countries-by-app-usage/

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI '10*, USENIX (2010), pp. 1-6.