Why Application Errors Drain Battery Easily? A Study of Memory Leaks in Smartphone Apps

Mingyuan Xia, Wenbo He, Xue Liu
McGill University, Canada
mingyuan.xia@mail.mcgill.ca
{wenbohe,xueliu}@cs.mcgill.ca

Jie Liu
Microsoft Research Redmond
jie.liu@microsoft.com

ABSTRACT

Mobile operating systems embrace new mechanisms that reduce energy consumption for common usage scenarios. The background app design is a representative implemented in all major mobile OSes. The OS keeps apps that are not currently interacting with the user in memory to avoid repeated app loading. This mechanism improves responsiveness and reduces the energy consumption when the user switches apps. However, we demonstrate that application errors, in particular memory leaks that cause system memory pressure, can easily cripple this mechanism. In this paper, we conduct experiments on real Android smartphones to 1) evaluate how the background app design improves responsiveness and saves energy; 2) characterize memory leaks in Android apps and outline its energy impact; 3) propose design improvements to retrofit the mechanism against memory leaks.

1. INTRODUCTION

According to the J.D. Power and Associates survey involving 7,000 smartphone customers [1], battery life has become a critical factor for user satisfaction. The energy-saving requirement has given rise to new innovations in mobile operating systems [13] and energy APIs [8] for apps. While proven to save energy for common mobile usage scenarios, these designs also arouse new application errors that cause abnormal energy drains [9]. In this paper, we make a step to answer the question: how a typical application error like memory leak impacts OS energy-saving mechanisms?

Most mobile OS inventions are driven by the observation of usage patterns. In particular, smartphone usage observes short app interaction time and frequent switches between different apps. According to a smartphone trace study [10, 12], 80% of mobile app usage ends within one minute. This short interaction time naturally requires short app loading time. Facing this demand, a mobile OS launches applications once and keeps them in the memory as long as possible. At any time, several *background apps* stay in memory while only one *foreground app* is interacting with the user. When the user switches apps, the OS can probably wake a background app and avoid loading it from scratch. According to our measurement, keeping apps in the background can reduce 60% to 90% of the overall app loading time and significantly prolong battery life.

To keep more apps in memory, individual apps should limit its memory usage. Hence, memory leaks that incur memory pressure can prevent the OS from keeping background apps. Consequently, lots of normal apps need to be reloaded over and over as long as only a few apps are leaking memory. To better understand memory leaks in smartphone apps, we study real world bug reports from Google Code and identify the memory objects that cause most memory leaks in Android apps and root causes for these bugs. Based on this knowledge, we propose to effectively tolerate memory leaks at runtime with low overhead by vetting most leaked objects on most likely leaking code paths.

Although memory leaks have been studied in the context of software and performance bugs [2, 6, 7, 5], our contribution is to how this kind of bugs causes battery draining problem (i.e., energy bugs) on smartphones, as extending battery life is first-class priority in mobile devices. We study memory leaks in the smartphone context to understand the root cause and design targeted and effective workarounds. Our study reveals new usage scenarios (such as screen rotation) that can trigger memory leaks and propose an approach that tackles these scenarios.

The remainder of this paper is organized as follows. Section 2 evaluates the background app design. Section 3 characterizes real-world memory leaks and relate these bugs to background apps and energy. Section 4 presents the related work and Section 5 concludes.

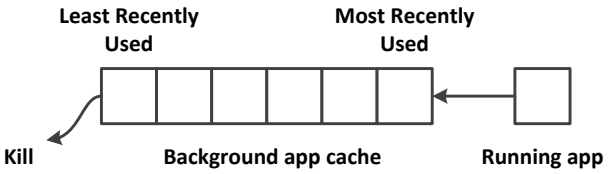


Figure 1: The fix-sized background app queue sorted in LRU order.

2. BACKGROUND APPLICATION CACHE

In this section, we explain background app mechanism, the important mobile operating system design that reduces app loading. We conduct several experiments on real smartphones to show how this mechanism saves loading time and energy.

2.1 Overview

On smartphones, loading an app normally involves several time- and energy-consuming phases (e.g., loading the executable content from storage, establishing initial network connections, etc). To improve app switch experience, all major mobile operating systems (iOS, Android, Windows Phone, BlackBerry) keep some non-running apps in memory. Next time these apps are launched again, the OS can wake the background app and reuse the old process (along with the executable code, file handlers, application data cache, etc).

Swapping Policy To avoid using too much memory, mobile OSes keep a fix-sized background app queue, as shown in Figure 1. Apps in this queue are sorted in LRU order and the (only) foreground app is always the most recently used one. Every time a new app is to be launched, the OS will check f it is present in the queue. If present, the OS will wake the background app and adjust the LRU sequence. Otherwise, the OS will replace the least recently used app with the app to be launched. In this case, launching app will involve app loading.

Although the queue is fixed in size, memory hungry background apps can still use up all system memory. Thus an Out-Of-Memory Killer (OOM) is introduced to kill background apps in LRU order when the system is about to running out of memory. As a result, the more memory individual background app uses, the fewer apps can stay in the cache.

2.2 Energy Impact

Experiments. If the app to be launched is in the background, this mechanism can save significant loading time (and thus energy). To evaluate the effectiveness of this mechanism, we conduct our experiments on a Samsung Galaxy Nexus S(i9100) smartphone installed with Android 4.03. Figure 2 lists the launching time for popular apps of representative categories. We monitor the CPU usage, IO activity and other resource usage (such as sensors) during app loading.

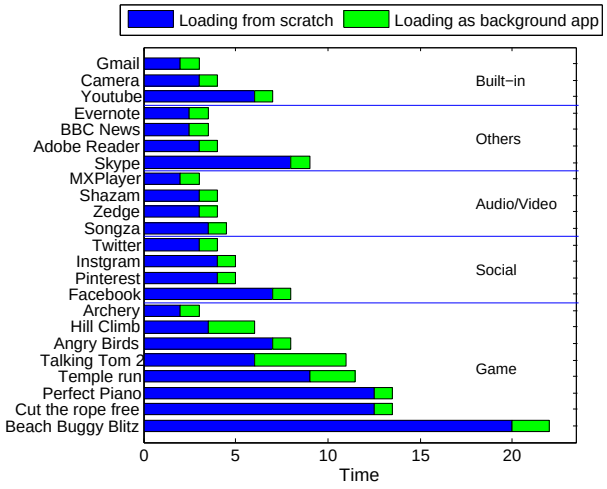


Figure 2: Compare the time for loading an app from scratch and waking a background app.

When loading, the app process (along with some helper processes such as the System UI process) occupies most CPU time. During this period, the screen backlight is on and causes the major energy consumption [4]. In addition, various IOs related to energy consuming components [4] also contribute notable consumption.

Storage IO. Loading an app incurs storage IOs to read app code and data. This typically involves several MB IO traffic (depending on the app size). According to our measurement, the IOs required to load an app are generally one or two orders of magnitude more than waking a background app. Particularly, as shown in Figure 2, games load much slower than other apps since they mostly contain large graphics data.

Network activity. Most network-based apps establish network connections on loading and reuse these connections afterwards. When an app becomes a background app, network connection states stay in memory and can be used on next launch. If the app has to be loaded from scratch, it requires extra TCP handshake and network activities to initialize the connections. For example, the skype app performs several rounds of coordinations with the server when loading. But once connected, the app only uses the network on demand.

Application data cache. Apps cache data (such as bitmap) in memory to reduce expensive IOs. When

Android (project name: android)

18273, 37607, 34731, 39821, 39819, 39818, 21189, 29306, 22794, 20724, 15170, 40552, 28524, 21965, 25442, 17903, 17015, 18001, 29884

Exotic Apps

Project name	Issue ID
anttek	2
mapview-overlay-manager	21
mapsforge	72
eyes-free	100
osmdroid	265
android-rcs-ims-stack	107
roboguice	102
mconf	251
robotium	331
libgdx	460
adwhirl	71
gmaps-api-issues	4766

Table 1: A list of memory leak bug reports. Available at `http://code.google.com/p/<project_name>/issues/detail?id=<issue_id>`.

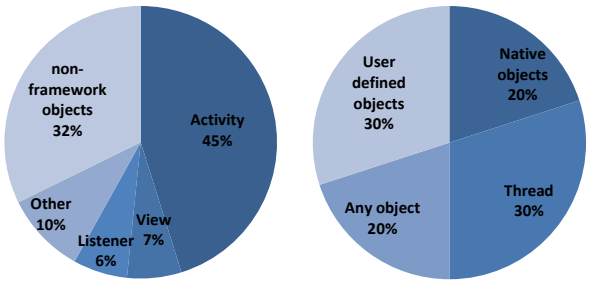
waking a background app, all the cached contents in memory can reused. On the contrary, a newly loaded app typically needs to issue IOs to fill this data cache. The most notable example is the built-in web browser. When kept in the background, the browser keeps all loaded web pages in memory. So by waking the background browser, the user can continue browsing without network activities. However, a newly launched browser needs to load the page and pictures. As a result, loading an app triggers IOs to fill application cache, which in turns drains the battery.

Overall. We conduct stress tests to reveal how these extra IOs from app loading impact battery life. In this experiment, apps are randomly launched until the battery is exhausted. We compare the battery lifetime when the background app mechanism is turned on/off[1]. The results show that the smartphone can last around twice longer if the OS can keep background apps. We attribute the reduced battery life to the combination of expensive IO operations described above.

3. MEMORY LEAKS IN APPS

The OS reduces energy consumed by app loading by keeping as many background apps as possible. However, given limited system memory, the more memory each app uses, the fewer background apps exist. In this section, we show that app memory leaks can cripple this mechanism and cause severe battery drain. To understand these bugs, we characterize real leak bugs in Android apps from Google Code. Then, we propose effective workarounds based on the knowledge gained from the characterization.

[1]On Android set Setting→Developer Options→Background process limit to zero to disable the mechanism



(a) Framework (five kinds) and non-framework objects.

(b) Breakdown of non-framework objects.

Figure 3: Breakdown of leaked objects.

3.1 Bug reports

We collect 31 bug reports (shown in Table 1) from projects hosted by Google Code. 60% of the bugs are found in app programming framework, libraries and built-in apps of the Android OS. The rest 40% are found in developers' apps. The studied apps fall into representative categories found in real app market. Some bugs found in gaming engines and testing frameworks can potentially affect a lot more apps. We study these existing bugs to understand three important facts: 1) what objects cause most memory leaks; 2) what are the common pitfalls that lead to memory leak; 3) how memory leaks affect background apps and energy consumption.

3.2 Objects with Most Memory Leaks

First, we want to identify what objects are most likely to be leaked, which helps to develop targeted methods. Figure 3 presents the distribution where objects from the Android programming framework are mostly leaked.

Activity and framework objects. *Activity* is the most leaked object in the Android OS as well as exotic apps. An activity is a programming component that manages the resources used by a window. When the associated window is exited, `onDestroy` method of the activity is called. `onDestroy` is supposed to remove all reference to the activity to allow it to be garbage collected afterwards. However, since lots of objects keep reference to the activity for system services, carelessly keeping the reference of a destroyed activity leaks its memory. Futher-

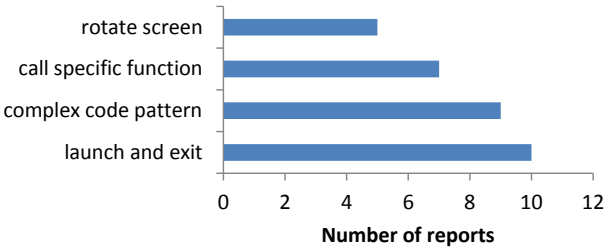


Figure 4: The breakdown of memory leak triggers.

more, most framework objects (such as Views and Listeners) indirectly reference the activity instance. So if any of those framework objects is leaked, the activity is also leaked.

Non-framework objects. Non-Android objects account for only 32% of the leaked objects. Unlike the situation for desktop applications [2], user-defined objects are not the major source of memory leaks (only 10%). This suggests that developers are more likely to misunderstand the life cycle of Android components.

From these observations, we conclude that a leak detector can focus more on activity instances. More specifically, the detector can report leak if an activity instance still exists while it should be recycled on current app state.

3.3 Leak Trigger

We identify that memory leaks are triggered by a few straightforward user interactions, as shown in Figure 4. These interactions normally stress certain app code paths which contain common programming pitfalls.

Launch and exit. Launching and exiting an app involve the life cycle methods of the main activity. When starting, the activity starts other program components, possibly working in an asynchronous way to acquire resources. On exit, the activity is supposed to release all the resources used by these asynchronous components (some might not finished yet). Failing to do so will lead to memory leaks. Previous literature [9] reports that apps that require locking hardware in high power mode can encounter malloc-without-free-like bugs. We find out that these bugs also leak considerable memory (due to leaking the activity instance).

Screen rotation. This simple user interaction can happen anytime during the activity execution. In response, the current running activity will be destroyed and a new instance will be created. During this procedure, certain program state objects from the old activity can be transferred to the new activity. However, if these state objects reference the old activity, the new activity will then indirectly keep the old one alive. To avoid memory leaks in this scenario, the programmers need to scrub the state objects being transferred. However, since screen rotation happens asynchronously, the state objects may contain some temporary objects that still hold the reference to old activity instance.

Calling a specific function. Some memory leaks only involve one function. Usually, after calling the function, the parameter objects will be referenced by some global variable and become not garbage collectable. The development document has recommended to use *weak reference* to allow JVM to recycle objects within global container. However this idiom is not commonly used (especially in exotic apps) or sometimes not properly used.

3.4 Energy Impact

Every time the user launches a leaking app, it leaks memory and becomes the most recently used app in the background app cache. Consequently, the leaking app keeps occupying memory until all other less recently used background apps are killed. In such case, the operating system is more likely to encounter memory pressure and kill some background apps. All the killed normal apps will have to be reloaded when the user launches them again, which incurs app loading overhead and energy consumption.

In short, leaking apps force the OS to waste time and energy reloading normal apps. The background app cache extends the life time of leaking apps and amplifies these bad effects.

3.5 Reinforcement

As a leaking app will become a long-term pain as long as it is launched once, we propose to retrofit the swapping policy to reduce the length of leaking effects. Our method aims to incur minimal runtime overhead as well as limited energy consumption. Our approach contains two modules: a lightweight leak detector telling which app is leaking memory and a priority adjustment module that prioritizes killing leaking apps.

Online leak detector. Existing leak detectors intend to detect all memory leaks. However, these approaches incur notable overhead (2x-100x runtime slowdown and 2x-10x memory footprint), which makes them only applicable during app development phase. We propose to monitor only activity instances, the most leaked objects according to our prior study. For an app being monitored, our detector records all live activities in the sequence of creation time. Then the detector calculates the mem-

ory used by long-living activity instances. If that exceeds a threshold, the app in question is marked as a potential leaker.

Background app priority adjustment. When a new app is launched, the mobile OS needs to update the background app queue. Instead of doing the LRU replacement, our adjustment module prioritizes killing leaking apps by adjusting its position in the background app cache. As a result, memory leaking apps will have a higher probability to be killed in face of memory shortage.

Performance Overhead. Since our detector only keeps track of activities, the memory footprint is proportionally to the number of live activities. Normal apps with only a few live activities will observe low memory overhead. The priority adjustment module calculates priority when replacing background apps, which is an O(n) list scanning. Overall, this detector can focus on most leaked objects while the adjustment module can prevent leaking apps from affecting other apps with an acceptable runtime overhead and memory footprint.

4. RELATED WORK

Energy bugs. Mobile OSes introduce new mechanisms to improve energy efficiency. Unfortunately, these designs also give rise to *energy bugs* [8], a type of programming errors that cause abnormal battery drain. Specifically, no-sleep bugs[9] causes heavy battery drain due to misuse of *wakelock*, a new energy API proposed by Android. Our work also investigate mobile OS energy-saving mechanisms. We evaluate the background app cache and show its effectiveness, as validated by other literature [13]. Meanwhile we show these new mechanism are somehow fragile in face of traditional programming errors, which now can lead to energy problems.

Leak detection and tolerance. Most memory leak detectors [2, 6, 7, 5] for desktop applications work in development phase and intend to report all potential leaked objects. Other runtime approaches [3, 11] aim to tolerate memory leak by swapping leaked object to secondary storage. However, storage IOs consume notable energy, which makes these existing approaches less applicable on smartphones. Facing these limitations, we propose our lightweight and targeted detector targeting only based on our characterization knowledge.

5. CONCLUSIONS

Battery life is a critical factor for user satisfaction of modern smartphones. In this paper, we first show how background app cache avoids repeated app loading and saves energy. Then we show that memory leaks can cause excessive memory consumption and cripple this mechanism. Our characterization study of real bugs shows that most bugs leak *activity*, the core Android app component. Also simple app state change such as frequently switching and screen rotation can trigger memory leak. Based on these observations, we design a lightweight leak detector that focuses on activity leak and a priority adjustment module to prioritize killing leaking apps.

6. ACKNOWLEDGEMENT

This work was supported in part by the NSERC Discovery Grant 341823. The authors also want to thank anonymous reviewer for their feedbacks and Yi Gao for discussions during the preliminary stage of the project.

7. REFERENCES

[1] J.d. power and associates reports: Smartphone battery life has become a significant drain on customer satisfaction and loyalty. `https://pictures.dealer.com/jdpower/166f0d180a0d02b7014443191870cdac.pdf`.

[2] Bond, M. D., and McKinley, K. S. Bell: bit-encoding online memory leak detection. In *ASPLOS'06*.

[3] Bond, M. D., and McKinley, K. S. Tolerating memory leaks. In *OOPSLA '08*.

[4] Carroll, A., and Heiser, G. An analysis of power consumption in a smartphone. In *USENIX ATC'10*.

[5] Hauswirth, M., and Chilimbi, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS'04*.

[6] Jump, M., and McKinley, K. S. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL '07*.

[7] Mitchell, N., and Sevitsky, G. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *ECOOP'03*.

[8] Pathak, A., Hu, Y. C., and Zhang, M. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *HotNets'11*.

[9] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys '12*.

[10] Shepard, C., Rahmati, A., Tossell, C., Zhong, L., and Kortum, P. Livelab: measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.* (2011).

[11] Tang, Y., Gao, Q., and Qin, F. Leaksurvivor: towards safely tolerating memory leaks for garbage-collected languages. In *USENIX ATC'08*.

[12] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M. How far can client-only solutions go for mobile browser speed? In *WWW '12*.

[13] Yan, T., Chu, D., Ganesan, D., Kansal, A., and Liu, J. Fast app launching for mobile devices using predictive user context. In *MobiSys '12*.