國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

Android 系統中應用程式即時回應性之分析

Anatomizing Application Responsiveness on Android

高全毅

Chuan-Yi Kao

指導教授：蘇雅韻博士

Advisor: Ya-Yunn Su, Ph.D.

中華民國 101 年 8 月

August, 2012

致謝

首先我要感謝我的指導教授蘇雅韻博士，如果沒有她的針砭和敦促，這篇論文不可能如期完成。同時我也要感謝兩位口試委員賜與我寶貴的意見，他們肯定了這個研究的價值，並且為這篇論文的後續研究方向給出指引。我還必須感謝實驗室的同學和學弟們對我這個好為人師的學長的容忍，尤其是唯一一個能和我討論安卓系統相關問題的郭嘉偉學弟。在這段研究的過程中若是連一個可以討論的人都沒有，我可能早就撐不下去了而放棄了。最後我要感謝我的家人以及摯友們，在我瀕臨瘋狂邊緣時，他們不離不棄的陪伴和無條件的支持，給了我排除萬難的勇氣。

Abstract

As users tend to judge the performance of a smartphone by interactive responsiveness, how various factors affect the perceived responsiveness remains unclear. In this thesis, we quantified the screen-switching responsiveness on Android smartphones by activity switch time for the first time. We compared activity switch time on different generations of hardware, different major versions of Android, for different implementation of similar functionalities and different previous state of the target activity when resumed. Our experiment result shows that while the response time to screen-switching actions is acceptable in average, the perceived performance is rather inconsistent. In the end, we suggest to consider activity usage pattern in memory management.

中文摘要

雖然使用者傾向於用操作時感受到的即時回應性來評斷一隻智慧型手機的效能，各種軟硬體因素是如何影響使用者感受到的即時回應性卻未見明朗。在這篇論文中，我們量測智慧型手機在真實使用狀況下切換畫面所花的時間來作即時回應性的分析。我們比較切換畫面所花時間，在不同世代的硬體及不同版本的安卓系統上的差異；同時也比較不同的應用程式對於類似功能的實做，以及目標畫面所屬程式的原始狀態，造成切換畫面所花時間的不同。我們的實驗結果顯示，儘管在當代的智慧型手機上，切換畫面的時間平均來說可以接受，但使用者卻可以感受到效能上的不穩定。最後，我們認為在記憶體管理的機制當中，若能參酌使用者對於畫面的使用習慣，可以更進一步改善使用者在切換畫面時感受到的即時回應性。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

On interactive computer systems, user satisfaction on computing performance is dominated by responsiveness. Previous studies showed users tend to evaluate the computing performance by comparing the length of response time to specific actions to their expectations [22]. Moreover, the expectation on response time is adaptively shaped by previous experience in performing the same action [22], depending on individual [11], user interface and task semantics [1]. For example, users are not be annoyed if it takes five seconds for a PC to wake from sleep, while they expect Microsoft Word to be launched around four seconds [1]. The variation of the response time also affects the perceived performance. Occasionally long response time make users think of whether the system is not functioning while surprisingly short response time may be considered a sign of wrong input [22]. In general, consistent, shorter response time leads to higher user satisfaction on the perceived computation performance despite potentially higher error rate in performing tasks [22].

As the above conclusion are drawn regarding desktop usage, on mobile devices similar correlation persists and the expectation on responsiveness is more stringent. It is common when a user wants to know if a smartphone provides good "performance", the user scrolls up and down through a web page, or taps to open multiple applications and switch among them, to see if the device can respond to these actions in a timely manner consistently as expected. In other words, the length and variation of response time to user actions is similarly indicative of the perceived performance of a smartphone. Additionally, the expectation on the length of response time is arguably shorter than in desktop settings because the length of individual interactive sessions are short on smartphones [13][10]. A recent study has already shown that the relatively long application launch time of a couple seconds can irritate users because users demand agility in mobile usage [**?**].

However, how the responsiveness of a smartphone is affected by hardware capability and software stack remains unclear. While various benchmarks are developed to help quantify the performance of smartphones performing specific type of computation such as graphic rendering and floating point calculation, it is not trivial how these benchmark scores translate into different quality of application responsiveness. For example, many users have reported that their smartphones are not running smoothly as expected while the benchmark scores are high compared to other devices with lower benchmark scores [25]. This particular scenario is probably raised by the throughput-oriented and task-based nature of these benchmarks. First, throughput benchmarks failed to capture the variation of response time to individual user action, while users feel bad about highly variant response times [12]. Second, throughput benchmarks typically drive the system by feeding user input as rapidly as the system can accept it, This behavior is equivalent to modelling an infinitely fast user with no think time regardless of how users really use their devices, as users typically wait for screen updates and think before taking their next action [12].

Furthermore, the influence of usage workload on responsiveness has not been explored. In addition to what we have stated above, task-based benchmarks and microbenchmarks also do not take into account the workload on a smartphone in real usage. We argue that usage has a significant influence on the responsiveness for the following reasons. First, different users have reasonably different set of applications and different usage pattern

[13], which result in diverse workloads of arguably different weight. Second, contemporary mobile operating systems such as iOS and Android enable multitasking which allows tasks to be executed in the background as the user performs foreground tasks [16][4]. Thus, to be realistic, the responsiveness of a smartphone cannot be comprehensively analysed without considering of real usage workload. Nonetheless, since for most real applications we have no access to their source code, it is difficult to reason about whether it's the application code or other factors that caused the observed responsiveness problem.

In this thesis we measure the responsiveness of Android smartphones by the response time to "screen-switching" actions, the activity switch time. We measured and analysed the activity switch time for a list of candidate activities during real usage, on top of a combination of different generations of hardware and different major versions of Android. We also compare the measured activity switch time for activities with similar functionalities from different applications. Among other observations, we found that while the overhead of activity creation depends mostly on application implementation, it generally dominates activity switch time once required. This result suggests further improvement in responsiveness can be achieved by considering activity usage pattern while reclaiming memory.

Chapter 2

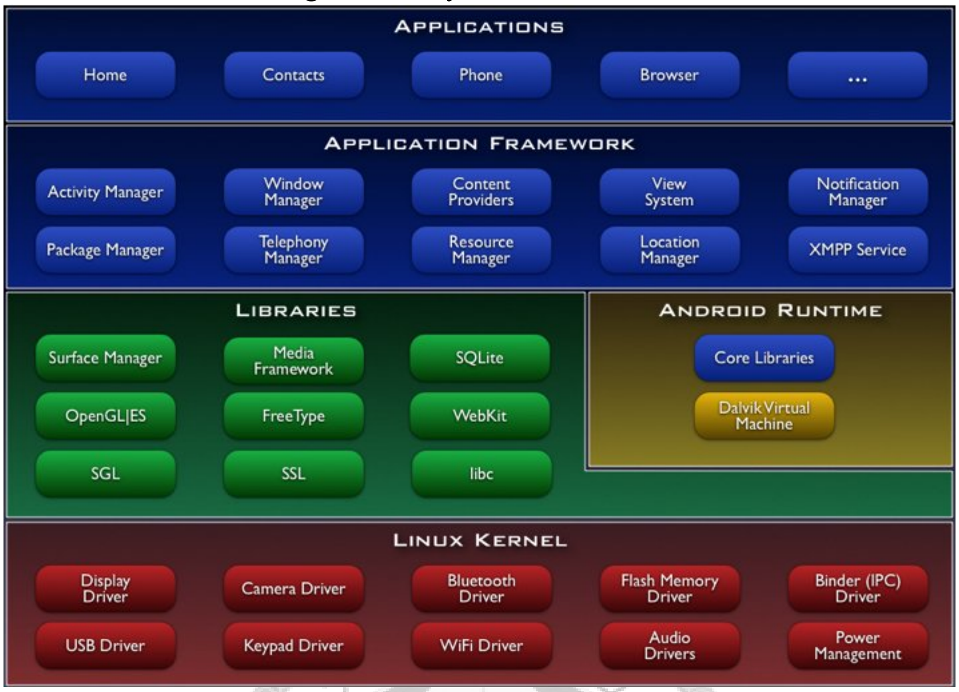
Background

This chapter provides necessary background for understanding the responsiveness of screen-switching events on Android. We start by giving an overview of the Android framework in Section 2.1, then describe the components an Android application is composed of in Section 2.2. The multitasking of Android is described in Section 2.3.

2.1 Android Framework

The software architecture of Android is illustrated by Figure 2.1. Android framework lies upon a trimmed Linux kernel customized to support only necessary functionalities for Android devices. On top of the kernel lies the hardware abstraction layer composed of hardware-dependent libraries. The core of Android runtime is the Dalvik Virtual Machine(DVM), which is an independent implementation of the Java language interface, as some of the code comes from Apache Harmony project [8]. Most of the framework components and services are written in Java, compiled into Dalvik bytecode and run in DVMs. For isolation, by default one application package runs in one instance of DVM in one process. Besides, each application package is assigned its own unique UID. As a result, one application cannot directly communicate with other applications without the help of the system.

Figure 2.1: System Architecture



2.2 Application Components

Android applications (apps) are generally programmed in Java language , optionally with some native components, using the API provided by underlying framework. On Android, apps are implemented modularly by extending the base classes described below [3].

Activity An activity is an component which controls a single screen to provide specific functionality [2]. An app usually consists of multiple activities that are loosely bound to each other. Typically there is an main activity to be launched when the application is first launched. Each activity can start other activities in order to perform other actions. For example, an SMS app might have one activity to show the conversation groups and another activity to compose an message. The user interface of an activity is typically defined using a hierarchy of views [6], of which the visibility are managed by the window manager service.

Service A service is a long-running component which runs in the background to perform work for remote components. Services are typically used to play music or fetch data from network in the background, without blocking user interaction with a foreground activity. A service does not have its own user interface but it can be started in other components such as activities and broadcast receivers. It is noteworthy that while iOS permits only predefined categories of background tasks such as music playback, Android allows arbitrary task to be performed in background services.

Content Provider A content provider manages a shared set of application data, which may be stored in the file system, an SQLite database, or any other persistent storage which your application has access to.

Broadcast Receiver A broadcast receiver responds to system-wide broadcast announcements originated by system or other applications. While broadcast receivers do not have their own user interface neither, they may create status bar notifications to alert the user when some event occurs.

2.3 Multitasking the Android Way

In this section, the multitasking mechanism of Android is described to illustrate how user tasks, and the switch between them, are handled on Android. How we define and measure the screen-switching time accordingly will be covered in detail in Section 3.1.

On Android, a user task is modelled by a sequence of actions, each realized by one activity component. Following this abstraction, users proceed from an activity to another to perform their tasks. As illustrated in figure 2.2, launched activities are organized into a "back stack", by which it means that when user navigates back by pressing the back key,

Figure 2.2: Task Management

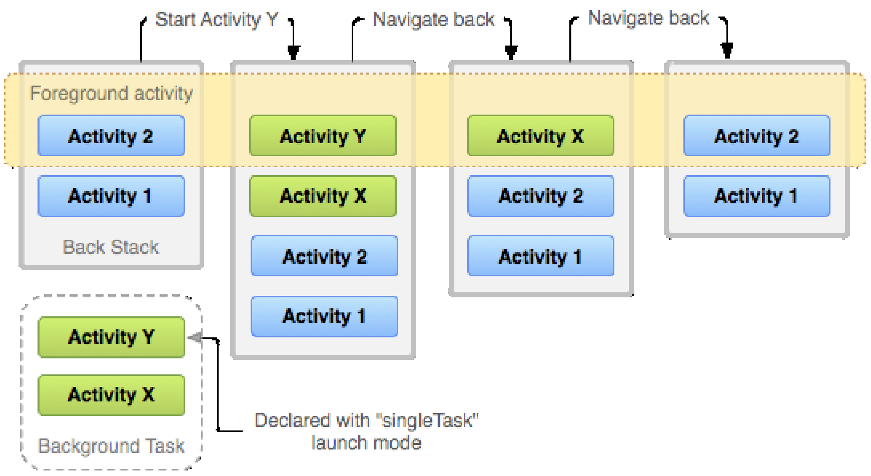


Table 2.1: Activity Lifecycle States

State	Receive User Input	Visibility
Resumed	Yes (focused)	Completely visible
Paused	No	Partially visible
Stopped	No	Completely hidden

the previous activity will be brought back onto foreground, popping the current activity out of the stack. The back stack is maintained by the activity manager service, which is a persistent system service responsible for managing tasks and processes.

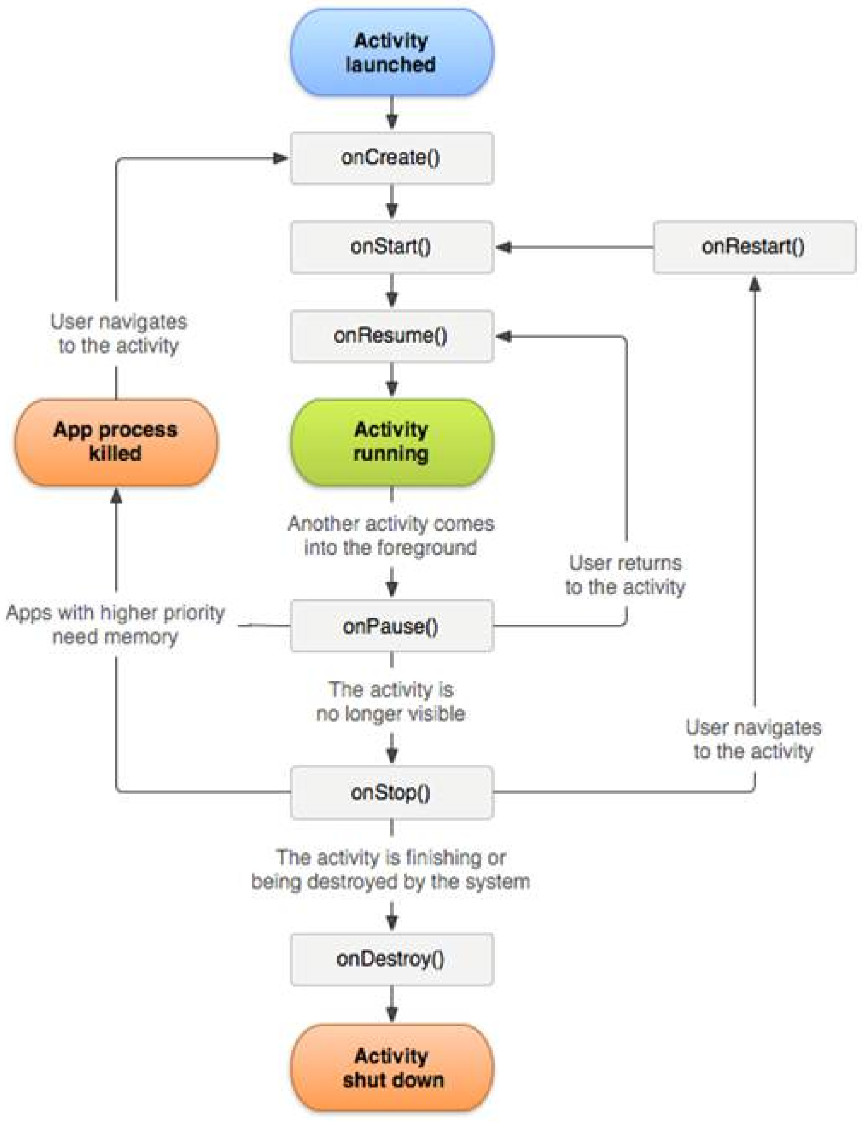
As activities are organized into tasks, the lifecycle of a single activity is defined upon its relation with other activities, with regard to the task it associates with. While only one activity can acquire user focus in the foreground at the same time, other launched activities are retained on the back stack, hidden from user. Accord to user focus and visibility, an activity can exist in three states as listed in Table 2.1. a more concrete definition of each state is as followed: A resumed activity is completely visible in the foreground and has user focus. This state is commonly referred to as "running". In this state the activity is responsible for responding to user actions. A paused activity is also visible in the foreground but without user focus. In this state the activity is considered alive and all the states associated with the activity are still maintained in memory. A stopped activity is completely obscured by other activities. A stopped activity is also considered alive as the activity object and its associated assets are still retained in memory.

When the user navigates to another activity, an activity switch is triggered, changing the visibility of the affected activities. As a consequence, the affected activities transition into and out of respective states and invoke corresponding callback methods as illustrated in figure 2.3. Applications can override these callback methods to get notified and do appropriate work when the state of the activity changes. If an activity is destroyed (removed from memory), when the activity is again launched, it must be created all over.

As activities transition through respective visibility states, the window manager service is notified to redraw the phone window to reflect the visibility change of affected activities. Since the target activity is supposed to resumed with its GUI brought to the front to received user focus, the window manager service calculates the expected visibility change of the GUI elements associated with the respective activities. In addition, Android by design enables applications to adaptively use different GUI assets when run on devices with different resolution in order to support various Android phones with different screen size and display resolution.

A unique aspect of the Android task model is that a task can span across multiple apps. For example, an app that needs to capture a photo can activate the Camera app, which will return the photo to the referring app. Nevertheless, since an activity cannot directly start another activity in other application packages. To enable this, an intent describing the intended action will be sent to the system whenever a user tries to perform another action. The package manager, which is responsible for keeping track of all installed application

Figure 2.3: Activity Lifecycle


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

packages, will resolve the intent and find the most suitable activity to launch for the specified action. As applications can also explicitly specify the target component and package name in the intent, an intent resolution is not always needed. In the end, a new process is created to instantiate the target activity if the process of corresponding application package is not currently running. In this way, Android handles the creation of processes and the instantiation of application components transparently.

In order to have shorter launch time, Android tries to keep an process in memory for as long as possible to minimize the need for process creation. The system kills old processes according to the relative priority of the process to reclaim memory for new or more important processes. The primary priority classes are described below:

Foreground The process holding the resumed activity is in this class. A running service which is temporarily set to foreground priority is also considered in this class. Foreground processes are have the highest memory usage priority because they are relevant to what the user is currently doing.

Perceptible A process in this class holds paused activity or perceptible background services such as music playback. Perceptible processes are also seldom killed because user may perceive the difference if they are killed.

Service Since Android implements priority inheritance, a service bound to higher priority components are elevated to the same priority of the binding component. The rest of the started services are in this class.

Hidden Processes holding stopped activities are in this class. Hidden processes can be killed without being noticed immediately by users. However, if the activity is needed again, process and activity creation will be required.

Empty Empty processes are not holding any application components and retained only for caching purpose, reducing the need for process creation.

Android starts killing processes of different priority categories when the amount of available memory becomes less than respective predefined value. Within each category the least recent used process is killed first. In this way the system can automatically select processes that least possibly affect user experience for reclaim.

Chapter 3

Methodology

3.1 Definition

Activity Switch Time (AST) In this thesis we leverage activity switch time to approximate the perceived responsiveness of a smartphone switching between screens of GUI. AST starts when the user expresses the intention of proceeding to another activity by touching screen or pressing keys, ends when the target activity is ready for the user to interact with.

Reason The reason of choosing AST as our measurement for responsiveness is threefold. First, in Android user tasks are modelled as a sequence of actions, each realized by one screen that is controlled by one activity component [5]. The event of switching activities naturally fits the screen-switching behavior, which users are used to leverage in judging the perceived performance of a smartphone. Second, given that a task may span across multiple applications and in-app activity switching is common, we can catch the interactive responsiveness of an Android smartphone more comprehensively in this way, compared to using the time needed to launch or switch between applications. Lastly, switching activity as a type of user action explicitly exposes the intention of user which provide clear semantics that enable better correlation with expectations on response time.

Limitation There are certain limitations with this definition. For one, most games are build upon proprietary game engines and libraries that simply do not follow the task model of Android. In this initial work we choose to base our analysis on typical and most general applications which follow the task model of Android. However, for these applications we also found that some of the them introduce an extra splash screen while loading the actual activity screen. In this case our measurement based on activity switch time may under-estimate the perceived responsiveness because splash screen may give user an impression that the application already starts responding. On the other hand, some of the applications finish the switching of activities early, showing stale content, and loading the latest content asynchronously in the background. In this case our measurement may over-estimate the perceived responsiveness because even the activity switch is considered done but the latest content is actually not yet ready for the user to interact with.

3.2 Measuring Activity Switch Time

Following our definition in previous section, AST is measured by the elapsed time between user expressing the intent of switching to another activity, and the target activity resumed in foreground with its screen ready for interaction. In this section we describe the activity switch procedure on Android in more detail, and how we extract the execution time breakdown. We divide the procedure into two parts, before and after system knowledge of an activity switch should be performed, for individual discussion in the following sections.

3.2.1 Triggering Switch

Input Event Dispatching When the user make input actions by touching screen or keys. the event is firstly captured by the device driver in the kernel. The input manager service of Android framework then deliver the event to corresponding application event handler according to focus state.

Input Event Handling The receiving application event handler calls to activity manager service once it determines that a switch to another activity should be done. Typically, the call includes an intent providing information about the intended action and a bundle of arguments for the target activity.

In this part we instrument the system code to keep track of the generation time of last detected input event. Once there is a call to start activity, the last input event is taken as the triggering event. In this way we can approximate the time elapsed between when user input is generated and when the system starts performing the resulted activity switch.

3.2.2 Performing Switch

Intent Resolution Once an activity switch request is sent to the activity manager service, it determines the corresponding activity component that should be resumed by consulting the package manager to resolve the intent. We measure the time spent in intent resolution by tracking the time spent in respective call to the package manager.

Task Management After the target activity component is determined, the system arranges the back stack, handles process creation and the instantiation of application and activity component if needed. In this part, we track every call to application and activity lifecycle callbacks and record the time spent. We also record the time spent in process creation to measure the overhead incurred.

Window Management When the task and process management is done, the window manager service is notified to redraw the phone window to reflect the visibility change of affected activities. Here we only record the event when the system finishes redrawing the phone window, as the end of our AST measurement.

3.3 Analysis Strategy

In this section we define the metrics and factors by which we try to analyse how these factors affect AST. In addition, we try to identify the most significant parts in the breakdown of AST.

3.3.1 Factors

In this thesis we try to analyse how different the AST of the same target activity will be on different generations of hardware model and different major versions of Android. We also compare the AST of activities providing similar functionalities but implemented by different application. In addition, since an activity can be started from different initial state and thus incur different amount of work as described in Section 2.3, we also compare the time needed to resume the same activity from different previous states.

3.3.2 Metric

Since users prefer shorter, consistent response times, following previous works [22][12], we consider the average length and coefficient of variation of AST to be relevant to user-perceived performance. The average AST of a specific target activity represents the over-

Table 3.1: Reference for user's reception on response time

Duration Range (ms)	Description
below 200	Instantaneous
500 to 1000	Immesiate
2000 to 5000	Continuous
above 5000	Captive

all impression on response time switching to the target activity. Seow et al. [20] provided a reference for how different length of response time may affect user's affluence in performing tasks, as shown in Table 3.1. Although they propose this reference for desktop use cases, we leverage it as a conservative guideline for mobile application responsiveness since no such reference for mobile use cases has been proposed. Meanwhile, the coefficient of variation resembles the consistency of response time whenever the user switches to the target activity. In this case, Seow [20] also suggests that users can perceive differences in duration range from 7% to 18% for durations up to 30 seconds.

3.3.3 Breakdown

In this thesis we try to break the AST of a specific activity into the four parts individually described in Section 3.2: handling input event, intent resolution, task management, and window management. Furthermore, we look into the execution time of individual activity lifecycle callbacks. We especially compare the process and activity creation time with regard to factors listed in Section 3.3.1.

Chapter 4

Experiment

4.1 Setup

To gain insight across different generations of hardware models and Android versions, in this thesis our experiments are conducted on combination of two generations of hardware along with two major versions of Android versions listed in Table 4.2. For hardware, the Google Nexus S (GT-I9020) is equipped with a 1 GHz Samsung Exynos 3110 CPU (ARM Cortex-A8) and 512 megabytes of RAM, while the Samsung Galaxy Nexus (GT-I9250) is equipped with a 1.2 GHz dual-core TI OMAP 4460 CPU (ARM Cortex-A9) and 1 gigabyte of RAM. The display is also different that the Google Nexus S has a 800×480 resolution while the Samsung Galaxy Nexus has a 1280×720 one.

For platform, Android 2.3.7 is the final version of the GingerBread branch while 4.0.4 is the lastest version of the IceCreamSandwich branch. We take the source code of both versions from Android Open Source Project [7] and build them using the default tool chain coming with the respective source tree. We also extract essential proprietary binaries from factory ROM images to make some of the hardware components work correctly such as the camera on Samsung Galaxy Nexus. Our candidate applications are either built-in or installed from Google Play. We assume these applications are optimized for respective hardware models and platforms since Nexus devices are intended to demonstrate the major release versions of Android. We list the class names and description of the candidate

Table 4.1: Candidate Activities

Tag	Activity Class Name	Functionality Description
Launcher	com.android.launcher2.Launcher	Default home launcher activity
Browser	com.android.browser.BrowserActivity	Default browser app: main activity
Camera	com.android.camera.Camera	Default camera app: main activity
AlarmClock	com.android.deskclock.AlarmClock	Default alarm clock app: main activity
Settings	com.android.settings.Settings	System settings: main activity
Phone	com.android.contacts.activities.DialtactsActivity	Default phone app: main activitity
Facebook	com.facebook.katana.activity.FbFragmentChromeActivity	Facebook: social news feed
Google+	com.google.android.apps.plus.phone.HomeActivity	Google+: social news feed
LineList	jp.naver.line.android.activity.MainActivity	Line: conversation list
WhatsappList	com.whatsapp.Conversations	Whatsapp: conversation list
LineSingle	jp.naver.line.android.activity.chathistory.ChatHistoryActivity	Line: single conversation
WhatsappSingle	com.whatsapp.Conversation	Whatsapp: single conversation

Table 4.2: Device Configuration

Tag	Hardware Model	Android Version	Number of Users
NS_2.3	Google Nexus S	2.3.7	1
NS_4.0	Google Nexus S	4.0.4	3
GN_4.0	Samsung Galaxy Nexus	4.0.4	4

activities in Table 4.1.

4.2 Data Collection

For data collection, we instrument the Android framework to record the necessary details in AST described in Section 3.2. We then give the devices to 8 users. These users are instructed to use the device as their main device for three to four weeks. Data are stored in the internal storage of the device until manual retrieval. The number of users of each device configuration is illustrated in Table 4.2.

4.3 Result

In this section we show our experiment results for how AST differs with respect to the factors described in Section 3.3.1.

Table 4.3: The average and CV of AST measured on three device configurations

Activity-Device	NS_2.3		NS_4.0		GN_4.0	
	Average	CV	Average	CV	Average	CV
Launcher	604.4	54.57%	483.1	78.44%	421.4	93.38%
Browser	888.9	64.22%	950.5	84.77%	790.1	95.79%
Camera			1542.9	18.68%	1242.2	34.54%
AlarmClock			475.6	91.37%	757.7	86.13%
Settings	496.3	39.36%	485.5	89.18%	599.1	52.26%
Phone			1305.9	69.63%	1129.0	68.29%
Facebook	2001.7	68.04%	3409.3	72.21%	1318.9	118.10%
Google+	503.2	44.74%	1077.6	116.20%	917.6	65.60%
LineList	835.8	80.38%	1092.0	102.52%	1279.6	76.19%
WhatsappList	474.2	52.62%	675.4	117.53%	717.5	87.11%
LineSingle	597.8	45.21%	626.6	98.79%	607.9	92.45%
WhatsappSingle	824.0	62.12%	964.2	72.13%	688.2	90.62%

4.3.1 Device Configuration

As we can see in Table 4.3, for Galaxy Nexus and Nexus S both running Android 4.0, 8 of our candidate activities have shorter average AST on Galaxy Nexus, which implies better overall speed in responding to activity switching actions from the users' point of view. For Galaxy Nexus, the larger display resolution requires more work to organize the GUI layout and redraw the phone window, while the far more powerful hardware sped up this operation. Meanwhile, for Nexus S running Android 4.0 and 2.3, for all our candidate activities the variation of AST on Android 4.0 is larger, which users may perceive as more unstable performance.

Overall, in 22 out of 33 cases we measured average ASTs of below 1000 ms, in the immediate range suggested by Seow [20]. In 8 of the cases the average ASTs lies between the immediate range and continuous range. Only in 2 of the cases the average ASTs reach the continuous range. However, as wee can see in table 4.3, for all the cases we measured a coefficient of variation of more than 20%, by far surpassed the perception threshold [20][11]. This result suggests that even though the response time of screen-switching actions on contemporary smartphones may seem to be acceptable from a lenient aspect, user can still feel the inconsistency of the perceived performance.

4.3.2 Application Implementation

When we try to compare similar functionalities realized by different apps we find the following results in Table 4.3. For single conversation screen, Line generally outperforms Whatsapp on all three device configurations. However, for listing conversations, Whatsapp in turn outperforms Line because the latter incorporates other functionalities in the activity showing list of conversations which burdens the resuming procedure. For showing social news feed, Google+ responds faster than Facebook. A possible reason is that Facebook leverages a embedded WebView browser which complicates the resuming procedure, while Google+ directly leverages platform APIs to realize its functionalities.

4.3.3 Program State

Since resuming an activity from different previous state requires different amount of work as explained in Section 2.3 and Section 3.2, here we try to explore more about whether it really makes perceivable difference and what are the most significant parts of work in terms of execution time. In Figure 4.1, 4.2, 4.2, 4.4, and 4.5 we classify our collected activity switching events into three classes: The class annotated by "proc" in the y-axis labels requires process creation (and thus activity creation). The second class annotated by "act" requires only activity creation. And and class annotated by 'none' requires neither process nor activity creation. Furthermore, we break AST into parts as described in Section 3.2: event handling, intent resolution, process creation, application/activity lifecycle callbacks, and phone window update.

Figure 4.1: Launcher: Program State vs. Device Configuration

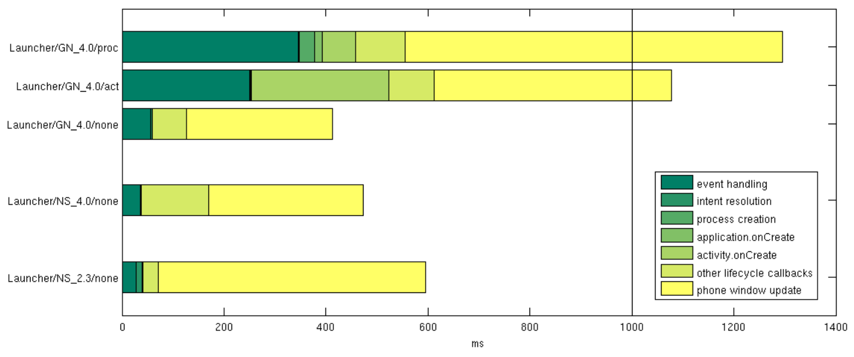
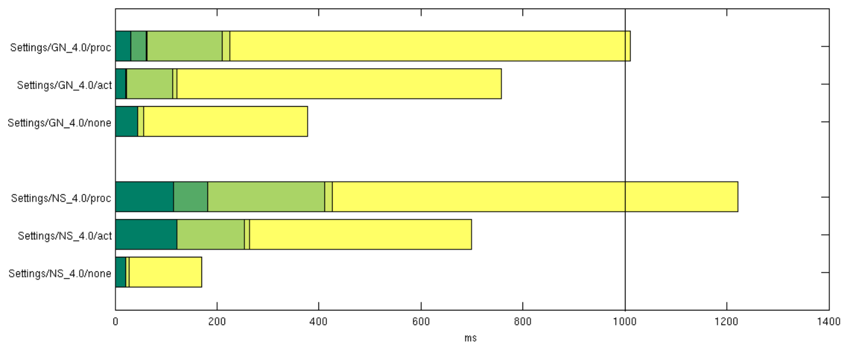


Figure 4.2: Settings: Program State vs. Device Configuration



As we can see in Figure 4.1, 4.2, 4.2, 4.4, and 4.5, the measured AST becomes significantly longer once process or activity creation is required for resuming the target activity. This observation agree with our knowledge in system mechanism that process and activity creation require respective per-package and per-activity instantiation and initialization. Also in these figures we can see that activity creation time dominates a significant part of AST once required, by far longer compared to process creation time. We can also see that the execution time of application/activity-on-create callback depends more on functionality and implementation of the target activity.

4.4 Discussion

In Android applications, activities typically load textures and image assets to set up content views in the onCreate method, which is called when the activity is created. Actually

Figure 4.3: Browser: Program State vs. Device Configuration

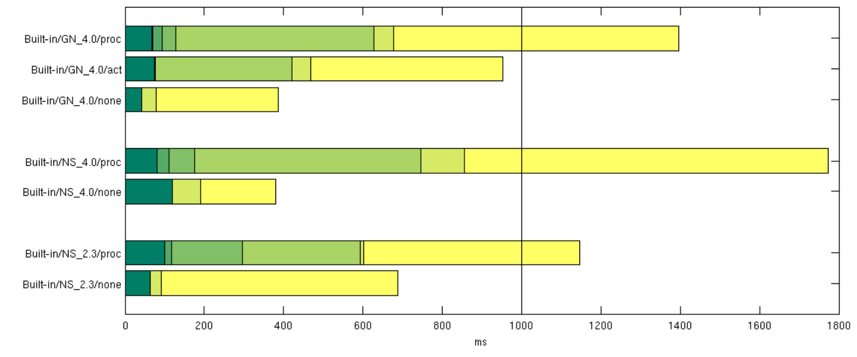


Figure 4.4: Facebook: Program State vs. Device Configuration

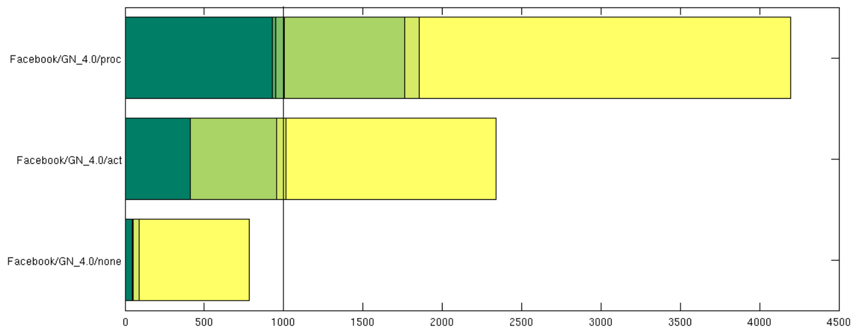
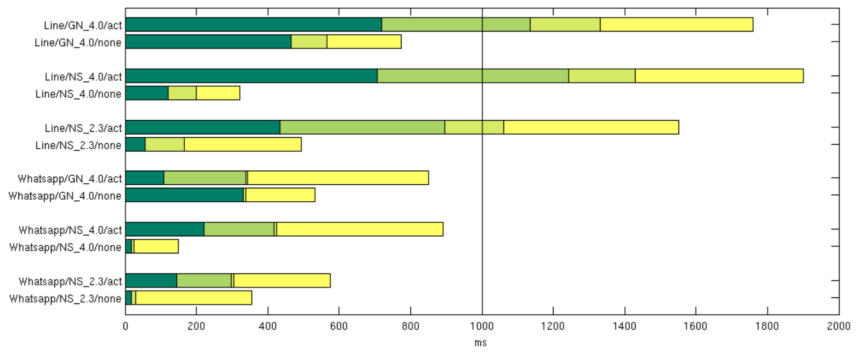


Figure 4.5: Conversation List: Program State vs. Device Configuration



in our observation, activity creation time dominates AST if exist. As a result, we suggest that reducing the need to recreate frequently-used activities may further shorten the average AST on a device. Furthermore, the benefit may be more significant on those devices with higher display resolution as the time needed to create activities and redraw phone window depends on the complexity of GUI layout and the size of image assets.

We also found that those applications without running background services are more likely to require process creation when switching to their activities. This observation follows that Android gives processes running services higher priority in memory residence than those only hold hidden activities. This policy makes sense assuming that services are running to do something relevant to user experience at the time, while hidden activities can be killed without being noticed. However, Android services can do literally anything for arbitrarily long unless getting stopped explicitly. This design effectively suggests application developers to implement unnecessary services to avoid the accommodating process being killed, occupying more memory and eventually result in more activity recreation while switching between activities. To cope with this problem we suggest to modify the existing mechanism to adaptively strike a balance in memory usage between background services and hidden activities with regard to the usage pattern of the user.

Lastly, we found that the event handling time spans across a kind of large range in our measurement. This result actually follows our measurement methodology that we include both the time spent in dispatching by the system and the handling by the previous activity. Since different activity can be launched from by a different set of previous activities, the time needed for previous activity to decide to switch to another activity is not all the same. We will try to further distinguish the time spent in future work.

Chapter 5

Related Work

responsiveness and user-perceived performance

The responsiveness of interactive computer systems has been established as an important indicator of user-perceived performance for long. In 1968, Miller first discussed the expected response time regarding a list of 17 situations, in the sense of conversational transactions between humans [18]. Miller stated that a computer system should take no more then 100 to 200 milliseconds to respond to a key-press or other similar action by the user. However, these requirements are established through his own beliefs and background on perceptual psychology.

Shneiderman (1984) first proposed defined an interactive model in which the response time of the computer system is defined as "the number of seconds that it takes from the moment a user initiates an activity (usually by pressing an ENTER or RETURN key) until the computer begins to present results on the screen or printer" [22]. In addition, Shneiderman stated that users evaluate the perceived performance of the computer system based on users' expectations on the response time of various actions. Schneiderman further discussed how users' expectation on response time depends on various factors such as task semantic, interface type, as well as the previous experience and adaptation of the users.

In 1996, Endo et al. indicated that the user-perceived performance of a interactive computer system is closely related to the response time [12]. Furthmore, he suggested that the user-perceived performance of an interactive system cannot be efficiently and correctly measured and analysed by throught-oriented benchmarks, and should be evaluated by the latency of processing individual user input events. In this sense, he proposed a way to capture the latency of procedding individual user input events by calculating the time between two subsequent CPU idle state.

Sebsequently, Seow provided a practical discussion of performance guidelines for software developers. In his work he proposed the 4 duration ranges within a typical span of immediate attention, and suggested that users can perceive the difference in performance if the difference in response time ranges from 7% to 18% for durations up to 30 seconds [20]. Based on Seow's recommendations, Anderson studied the relationship between the level of user satisfaction and different levels of computing performance in terms of the completion time of specific user task elements [1]. Anderson's result may serve as a verification that Shneiderman's statements mentioned above are true.

The relationship between responsiveness and user-perceived performance seemed well-studied in desktop environment. However, no previous work have discussed how user's expectations on responsiveness will change in a mobile environment. Since mobile intersections are short and users generally expect agility in mobile usage contexts, we believe that users expect even shorter response time while performing the same task such as launching applications. In the present study we leverage this belief as our assumption and explored how the responsiveness is affected by hardware, software, and usage characteristics on smartphone.

contemporary responsiveness problems and solutions

In desktop environment, solutions have been proposed to identify responsiveness problems. Basu et al. finds the processes that most probably caused a sudden slow down on a Windows PC by modelling the "normal state" of every process [9]. They use performance counter values as features such as I/O bytes read per second, percentage of CPU used, page faults, and so on. Meanwhile, Wang et al. pointed out a pattern that most responsiveness bugs, other than correctness bugs such as deadlocks, are triggered by invocation of blocking calls from the system UI thread, which should never be blocked [23]. In this sense, they proposed a static analysis framework that can find and help cure potential responsiveness problems in application code. While these approaches are established in desktop setting, they can also be applied in mobile setting to detect the same problem.

On mobile devices, the responsiveness problem has also been explored from various aspects. Among all the factors, the performance of wireless network has always been regarded as the main source of delay in network applications. Huang et al. anatomized the performance difference of web browsing on 5 different devices, with regard to different 3G network providers [15]. They measured the page downloading time to approximate the user-perceived response time in loading a web page, and concluded that network performance dominates the user-perceived performance of network applications such as browsers. Wang et al. further anatomized the performance of WebKit browser by measuring the detailed execution time breakdown in loading web pages [24]. Through dependency timeline characterization and what-if analysis, they concluded that resource loading is indeed the bottleneck step in loading web pages from wireless networks, which implicitly confirmed the previous statement made by [15].

Recently, more solutions have been proposed to help improve the interactive responsiveness on the relatively resource-constraint smartphones. Raghaven et al. tried to solve the problem by revision of hardware architecture [19]. Based on the assumption that most mobile applications do not demand sustained performance, they allow their chips to temporarily exceed the sustainable thermal power budget to do sub-second burst of intense parallel computation. Kim et al. studied the influence of different storage hardware on application performance [17]. They pointed out that the performance of underlying flash storage can indeed be a serious bottleneck for application performance against general intuition, due to the prevalent use of synchronized random writes through SQLite. Interestingly, they also identify that the application launch times do not significantly change even when all data is being read from memory, and suggests that storage is likely not a significant factor in application launch performance. At the same time, Yan et al. stated that the application launch time is too long for users to enjoy agility in mobile usage context [**?**]. They proposed a novel method to predictively prelaunch applications based on the history of application launch sequence with regard to time and location context. While they effectively shortened the average launch time of applications, they also admit that predictive prelaunching can result in variable perceived launch time which may hinder usability.

Above all, to our knowledge, we are the first to use activity switch time as a metric for interactive responsivenss on Android smartphones. We are also the first to measure and analyse data from real usage instead of controlled in-lab experiments. In this way we can effectively take the workload characteristic into consideration in our analysis.

smartphone usage trace collection and analysis

Since smartphones have become more and more prevalent, how users are using these devices in their own context, has become the research interest of vendors and developer communities.

In 2005 when the concept of smartphone had not emerged, Demumieux et al. designed and implemented a tool to collect usage data on mobile devices powered by Symbian and Windows CE. Their tool can be used to gather the relative frequency and time spent using each functionality. Besides, they also gathered the navigation behaviour, by their definition, the sequence of UI components the user interacted with. Subsequently in 2007, Froehlich et al. proposed a framework for capturing both objective and subjective in situ data on mobile devices [14]. Their tool sampled user experience by triggering surveys by sensor values.

In 2010, Falaki et al. conducted a comprehensive study of smartphone usage on Android and Windows phones, and demonstrated a way to accurately predict future energy drain based on the battery drain in preceeding time windows [13]. They found that despite quantitative differences, qualitative similarities exist among users. First, the relative usage frequency of applications for each user follows an exponential distribution. Second, the time between usage sessions can be captured using the Weibull distribution.

In the mean time, Shepard et al. presented a methodology to collect smartphone usage with a reprogrammable in-device logger designed for long-term user studies [21]. They were the first that deployed such a framework on iOS through jailbreaking. They also discussed the performance impact introduced by their framework in terms of power consumption. Later in 2011, Bohmer et al. presented a large-scale deployment based resarch study on Windows phones, which aims to derive more insight on how users are using their devices, and apps, with regard to location and time context [10].

While all previous studies agreed that smartphone usage behaviour differ among users, qualitative similarities were also identified such as the relative popularity of applications follows exponential distribution, and the usage behaviour sticks to diurnal pattern. However, none of the previous studies described above addressed the implication of usage pattern characteristics on the perceived performance of smartphones. In the present research we incorporate this aspect to make our analysis more realistic and more indicative of the performance perceived by individual users.

Chapter 6

Conclusion

In this thesis, we quantified the screen-switching responsiveness of applications on Android smartphones by measuring activity switch time for the first time. We compared how activity switch time differs with regard to different generations of hardware, different major versions of Android, different implementation of similar functionalities, and different previous state of the target activity when resumed.

In general, more powerful hardware provides shorter AST. However, a screen with higher resolution may introduce extra work in creating activity and redrawing phone window. Meanwhile, AST of the same activity appears to have higher variation on Android 4.0 than on Android 2.3. As our measured AST appears acceptable by users in terms of average length, the variation is high enough to be noticed by users, affecting the perceived performance thereafter.

As we found that activity creation time dominates the whole AST once required, we suggest to consider activity usage pattern in memory management. We leave the improvement of out measurement methodology, further modification of the system, and the respective evaluation as future work.

Bibliography

[1] Anderson, G., Doherty, R., and Baugh, E. Diminishing returns?: revisiting perception of computing performance. In *Proceedings of the 2011 annual conference on Human factors in computing systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 2703--2706.

[2] Activities. http://developer.android.com/guide/components/activities.html.

[3] Application fundamentals. http://developer.android.com/guide/components/fundamentals.html.

[4] Multitasking the android way. http://android-developers.blogspot.tw/2010/04/multitasking-android-way.html.

[5] Tasks and back stack. http://developer.android.com/guide/components/tasks-and-back-stack.html.

[6] Ui overview. http://developer.android.com/guide/topics/ui/overview.html.

[7] Android open source project. http://source.android.com.

[8] Apache harmony™ - open source java™ se. http://forum.xda-developers.com/.

[9] Basu, S., Dunagan, J., and Smith, G. Why did my pc suddenly slow down? In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques* (Berkeley, CA, USA, 2007), SYSML'07, USENIX Association, pp. 4:1--4:6.

[10] Böhmer, M., Hecht, B., Schöning, J., Krüger, A., and Bauer, G. Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In

Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (New York, NY, USA, 2011), MobileHCI '11, ACM, pp. 47--56.

[11] Dabrowski, J. R., and Munson, E. V. Is 100 milliseconds too fast? In *CHI '01 extended abstracts on Human factors in computing systems* (New York, NY, USA, 2001), CHI EA '01, ACM, pp. 317--318.

[12] Endo, Y., Wang, Z., Chen, J. B., and Seltzer, M. Using latency to evaluate interactive system performance. In *Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), OSDI '96, ACM, pp. 185--199.

[13] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 179--194.

[14] Froehlich, J., Chen, M. Y., Consolvo, S., Harrison, B., and Landay, J. A. Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *Proceedings of the 5th international conference on Mobile systems, applications and services* (New York, NY, USA, 2007), MobiSys '07, ACM, pp. 57--70.

[15] Huang, J., Xu, Q., Tiwana, B., Mao, Z. M., Zhang, M., and Bahl, P. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 165--178.

[16] ios: Understanding multitasking. http://support.apple.com/kb/HT4211.

[17] Kim, H., Agrawal, N., and Ungureanu, C. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 17--17.

[18] Miller, R. B. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I* (New York, NY, USA, 1968), AFIPS '68 (Fall, part I), ACM, pp. 267--277.

[19] Raghavan, A., Luo, Y., Chandawalla, A., Papaefthymiou, M., Pipe, K. P., Wenisch, T. F., and Martin, M. M. K. Computational sprinting. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2012), HPCA '12, IEEE Computer Society, pp. 1--12.

[20] Seow, S. C. *Designing and Engineering Time: The Psychology of Time Perception in Software*, 1 ed. Addison-Wesley Professional, May 2008.

[21] 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. 38*, 3 (Jan. 2011), 15--20.

[22] Shneiderman, B. Response time and display rate in human performance with computers. *ACM Comput. Surv. 16*, 3 (Sept. 1984), 265--285.

[23] Wang, X., Guo, Z., Liu, X., Xu, Z., Lin, H., Wang, X., and Zhang, Z. Hang analysis: fighting responsiveness bugs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 177--190.

[24] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2011), HotMobile '11, ACM, pp. 91--96.

[25] Xda-developers. http://forum.xda-developers.com/.