A Recipe for Responsiveness

Strategies for Improving Performance in Android Applications

Elin Nilsson

Elin Nilsson
Spring 2016
Master Thesis, 30 hp
Supervisor: Håkan Gulliksson
External Supervisors: Marcus Forsell Stahre, Keith Mitchell
Examiner: Thomas Mejtoft
M.Sc Interaction Technology and Design, 300 hp

Abstract

Mobile applications are expected to be fast and responsive to user interaction, despite challenges mobile platforms and devices face in terms of limited computational power, battery, memory, etc. Ensuring that applications are performant is however not trivial, as performance bugs are difficult to detect, fix, and verify. In order for mobile applications and devices to appear perfectly responsive to the user, they need to meet a 60 frames per second frame rate, and keep load times preferably between 0-300 ms. Meeting these expectations means that there is no room for performance bugs, so there is a need for improving and developing better testing tools and strategies in order to help mobile developers improve performance in their applications.

This thesis investigates strategies for testing and improving performance in Android applications by conducting a literary study, and a case study with the Spotify Android application. Some of the key findings of this thesis include promising results from tools that visualise sources of performance bugs in the user interface of applications, as well as proposed strategies and tools aimed to help developers profile and improve performance in their Android applications.

Acknowledgements

For their aid and encouragement throughout this thesis, and for ensuring that this report is as enjoyable, educational, and exciting as it ever could be, I would like to extend my gratitude;

To Marcus Forsell Stahre, for sharing your knowledge, and for your encouragement; To Sergey Basheleyshvili and Alexander Osmanov, for providing laughs precisely when I need it, and teaching me about your Russian ways; To Keith Mitchell, for your support of this project; To the GLUE Team, for taking me in with open arms; To Andreas Grünewald, for sharing this journey with me; To Arvid Bräne, Victor Winnhed, and Albin Hübsch, for your invaluable feedback and for taking the blame for any grammatical error that resides within these pages.

For the guidance, mentorship, friendship, and inspiration the following individuals have given me through the years;

To my mother and father, for supporting and loving me despite of me; To my brother, for being exactly you; To my family, for the never-ending support and for keeping me grounded; To Håkan Lenneby, for your enthusiasm and contagious love for education and science; To Thomas Mejtoft and Håkan Gulliksson, for your dedication to our program, and for your belief in what we do; To ID11, for everything that I cannot express in words; To ID, through and through, for being the best bunch of people.

And finally, to you, dear reader, for your interest. I wish you a pleasant read.

Have a lovely day.

Contents

1 Introduction

When smartphones first appeared on the market, they introduced new challenges for developers to consider: limited computational power, limited power supplies, fluctuating access to networks, limited screen size, portability, etc. Mobile applications are despite these challenges expected to be fast, responsive, and performant: slowness often gets brought up in application reviews and bug reports [40]. Generally, anything slower than 100-200 ms can be perceived by the human brain as explicit slowness or lag [17], which is a challenge for mobile developers since performance is a key aspect in a good user experience [51].

Software performance, in terms of code efficiency, execution speed, and perceived responsiveness, is one of the more abstract parts of computer science. Often, performance is in the eye of the beholder, and research has shown that performance bugs are more difficult to detect, test, reproduce, and fix than non-performance or functional bugs [47, 28, 36, 55, 45]. Mobile platforms pose new challenges where the context of use can greatly affect performance, making detection and reproduction of bugs even more difficult. What is making the application slow? Is it implemented inefficiently, is it running on an old device, does the user have poor reception or a bad network connection, or is there some other application draining all the resources of the device in the background? Although software performance is a rich research area, it is unclear whether that research is applicable for mobile software performance [45]. The new constraints introduced by mobile platforms require new research efforts to understand implications regarding performance, and there is also a clear need for more research about tools, guides, strategies, and support for mobile developers to improve performance [54, 49, 44, 45].

Android is since 2010 the biggest mobile platform in the world [35], with more than one million new devices being activated every day [9]. Part of the success of the Android platform is likely due to its open nature: the Android operating system is an open source project, and all the tools needed to develop applications for the platform, e.g. the Android SDK (Software Development Kit) and Android Studio, the official IDE (Integrated Development Environment) for Android development, can be downloaded for free on the Android developer website[1]. Due to the open nature of the platform, developers can use, override, or modify almost all system components, protocols, and APIs (Application Programming Interfaces), more so than on more closed platforms, such as Apple's iOS. This openness makes the Android platform a good research object, as the readily available source code enables developers and researchers to understand the platform better.

[1] `http://developer.android.com/sdk`

1.1 Spotify

Spotify is a music streaming company founded in 2006 by Daniel Ek and Martin Lorentzon. Spotify has over 100 million users, over 30 million of which are premium users, meaning they pay a monthly subscription rate. Spotify is available on multiple devices, including desktop, web, mobile, chromecast, smart TVs, as well as integrated in some cars and sound systems. The service is split into a free tier and a premium tier, where the free tier has limited functionalities and is supported by ad revenue and upsell to the premium tier. As of June 2015, Spotify had paid over $3 billion to right-holders, record labels, and artists, the platform hosts over 30 million songs, and the service is available in 59 different markets [52].

During the summer of 2015, the author of this thesis was an intern at the Spotify headquarters in Stockholm, and worked as an Android developer. During the internship, she developed tools integrated in the internal versions of the Spotify Android application. These tools could be used to debug user interfaces, both from a design perspective, and a performance perspective. These tools are introduced and described in chapter 4. This thesis is inspired by aspects of that work, and focuses further on strategies for improving performance.

1.2 Objective

The objective of this thesis is to research performance on the Android platform, and investigate the possibilities to contribute with tools and guidelines for Android developers to detect performance bugs in their applications. The scope of the thesis includes a literary study, an exploration of the Android platform, a case study with the Spotify Android application (hereafter referred to as the Spotify application), as well as evaluating and developing tools integrated in it.

This thesis aims to answer the following research questions.

- What are performance bugs, and how do they impact mobile development and platforms?
- What causes performance bugs in Android applications?
- Can causes of performance bugs be detected visually? Can visualisation be a valid strategy for performance testing and tooling?

1.3 Outline

The report continues with a chapter about related research, which introduces the topics of mobile software performance, and strategies for testing and fixing performance bugs. This is followed by a chapter about the Android platform, which introduces characteristics that affect performance. The next chapter presents the tools developed prior to this thesis, followed by a presentation of the methodology of the thesis work, which in turn is followed by a presentation of results and findings.

The report concludes with a discussion about the results, the thesis work, challenges, limitations, and future work.

2 Performance Research

This chapter introduces research areas related to this thesis, reviewed as a part of a literary study that will be presented in section 5.1. Notably, some of the most relevant and useful articles cited below stated that their research and studies were the first of their kind, to the best of their knowledge [47, 28, 45, 38, 44, 36, 46], indicating that this is an interesting albeit sparse area of research in need of contribution.

2.1 Performance Bugs

A software bug is an error or mistake in a program or system that causes it to crash, behave in unintended ways, or produce incorrect results. A performance bug is a type of software bug that causes execution times to be longer than they could be, and fixing the bug can significantly speed up software without altering its functionality [36]. Compared to non-performance bugs, which usually cause deterministic problems or failures, performance bugs are much more difficult to detect, fix, and test [47, 55, 36, 45]. Nistor et al. [47] found that fixes for performance bugs were generally larger, took longer, and needed more developers than those for non-performance bugs, and that any supplementary fixes to the bugs usually further improves performance, indicating that it is difficult to verify fixes as well as bugs. Zaman et al. [55] found that some performance bugs are intentionally left unfixed due to trade-offs between tolerable performance decrease and the difficulty level of fixing the bug.

Despite being difficult to detect, there are many studies about causes for performance bugs. There are several patterns for potential causes, for instance those identified by Jin et al. [36]: inefficient call sequences, functions doing unnecessary work, and synchronisation issues, the latter of which has been found to occur even if there is just one or two threads running [46]. Patterns specifically related to the Android platform were found by Liu et al. [45]: heavy operations on the main thread, computations for GUI (graphical user interface) elements that are not visible, and frequent, heavyweight callbacks, especially callbacks invoked by user interaction or changes in the activity lifecycle. For Android, there are four categories of API calls that are known to be heavy operations: network access, flash storage access, database access, and bitmap processing [54, 49]. If any of these operations are run on the main thread, they are likely to block UI (user interface) updates, causing stutters or “jank”.

2.2 Strategies for Testing and Fixing Performance Bugs

The main problem with detecting performance bugs is usually not that potential causes are unknown, rather it is the lack of testing strategies, especially for Android [54]. In general, there exist many testing tools and strategies for mobile development [42], as well as research and development of more such tools and strategies [25, 34, 2, 1, 3, 29, 30], although almost all are focused around non-performance bugs. This is natural considering the differences in nature between these types of bugs; non-performance bugs will usually cause an application to either crash or throw exceptions, which can be detected fairly easily through monitoring application behaviour, or analysing logs and stack traces.

However, studies like Liu et al's [45], where they found that over 11,000 out of 60,000 Android applications had suffered or were suffering from performance bugs, have inspired new strides in research around mobile performance bugs and testing strategies [48, 44, 54, 49]. One strategy focuses on detecting usages of code patterns known to cause performance problems, and developing tools to do just that. This strategy is based on the widely used concept of static analysis and rule based detection of known bug patterns [33, 37], used in tools like FindBugs [31], which analyses code and warns about usages of patterns that are known to cause bugs. Liu et al. [45], Jin et al. [36], Ongkosit et al. [49], and Jovic et al. [39] have all developed tools based on this strategy, all of which were found to show real promise in detecting performance bugs. Another strategy is to add artificial pauses to known blocking operations to reveal if those operations are actually causing blockages or not, which has been explored by Nistor et al. [48], and Yang et al. [54]. There are also tools and guidelines developed by Android and Google to help developers build performant applications, some of which will be presented in chapters 3 and 5.

Although promising, there are challenges with mobile development practices that might hinder the success of such tools and practices. There have been several studies identifying such challenges [50, 27, 43], as well as how developers work with testing [38, 41] and improving performance [44]. The findings of these studies includes challenges with cumbersome tools, poor documentation, time constraints, and lack of expertise [41], which causes many developers to prefer manual testing [38, 41, 44]. Johnson et al. [37] found that this was also the case for static analysis tools, one reason being their tendency to produce false positives. For new tools and strategies to be successful, they would need to take such challenges into account: be efficient to use, and easy to learn and set up, without adding too much overhead for new users.

2.3 Strategies for Visualising and Understanding Performance Bugs

Researchers have identified the need for improved detection measures for performance bugs, without producing false negatives, as well as support for fixing them [44, 45, 54]. Interesting measures towards meeting such needs include a tool developed by Beck et al. [26], which used in situ visualisation to augment performance data into code editors. The tool aimed to improve psychological motivation and avoid split-attention effects. Another interesting measure is a feature in the tool that

Jin et al. [36] developed, which in addition to warning about performance problems, false positives, and bad practises, also identifies good performance practices. These approaches could not only make it easier for developers to detect performance bugs, but also to understand causes for them, making it easier to fix them.

These approaches have inspired this thesis work, which aims to find strategies for detecting performance bugs that can help developers to better understand the causes of such bugs.

3 Android

This chapter introduces core concepts and components of the Android platform, focused on facets of the platform that affect performance. Most of the material covered is provided by Google through the Google Developer website[1], and in recorded talks and videos from the Google Developer YouTube channel[2].

3.1 Fundamentals of the Android Platform

Android is a mobile operating system based on the Linux kernel, and is mainly developed by Google [7]. The first version of the Android SDK (Software Development Kit) was released in 2008. The Android source code is released under open source licenses, and is written mainly in C, C++, and Java. The software stack consists of the Linux kernel, a Hardware Abstraction Layer, libraries like OpenGL|ES and SQLite, the Android Runtime, which consists of core libraries and the virtual runtime, the Android application framework, and then applications on top, as illustrated by Figure 1.

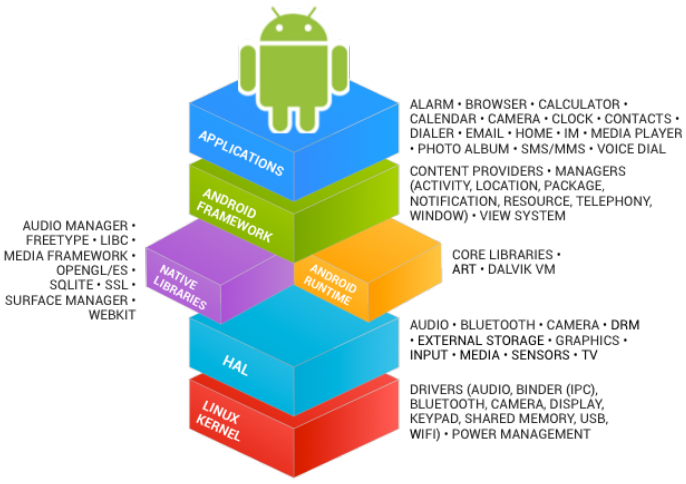


Figure 1: An illustration of the Android software stack, which includes the Linux kernel, a Hardware Abstraction Layer, third party and core libraries, the Android application framework, and the application layer [8].

Android applications are built using four different application components, namely activities, services, content providers, and broadcast receivers. Activities are usually the main building block of any application, as it is the sole component that can

[1] `http://developer.android.com`
[2] `http://youtube.com/GoogleDevelopers`

provide a UI (user interface) [4]. An activity follows a lifecycle: it is paused and resumed if operations outside the application occurs but the activity is still visible, stopped and started as the user is switching between applications, and destroyed and recreated if the system needs resources elsewhere. Services run in the background, independent of whether the application is in focus [23]. Content providers manage shared application data [11], and broadcast receivers allow for broadcasting events within or across applications [10]. All components are run by default on the main thread of the application.

3.1.1 Graphics and User Interface

The Android platform is normally based on a direct manipulation system, meaning that a touch screen is the main input for user interactions. Most devices have some hardware buttons as well, but generally, applications will rely on interactions with the UI. This section will introduce how the Android UI system works and how it is rendered.

UI Components

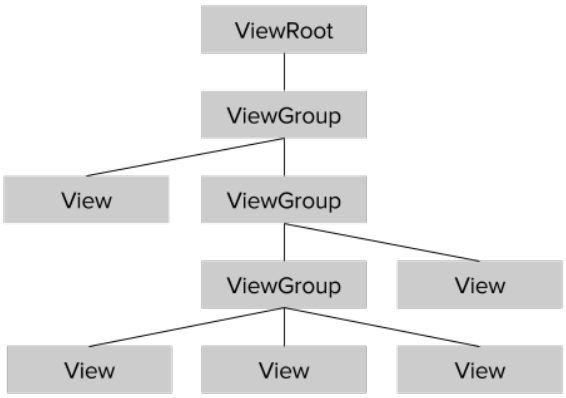


Figure 2: An example of a view hierarchy, where the ViewRoot is the root node, and other nodes are either Views or ViewGroups.

On Android, the GUI (graphical UI) consists of objects called views, which extend the View class, a class that implements logic for how the view is rendered and displayed on the screen. A view is a graphical object that controls a certain area of the screen, and can respond to user interaction. A view can either be a widget: a visual element such as a button, an image, or a text field, or it can be a layout: a container for other views. Layouts, which extend both the ViewGroup class and the View class, implement logic as to how its children are displayed and ordered. Layouts can be nested in other layouts, in which case they act both as views relative to their parent layout, and as layouts relative to their child views. The Android platform provides a UI Toolkit with a set of common layouts and widgets, such as buttons, text fields, images, grid layouts, list layouts, etc. Custom layouts and widgets can be created by subclassing a View or ViewGroup class, or any other

class that subclasses those. The UI can be declared through XML elements, or instantiated programmatically [18]. Figure 2 depicts an example of how widgets and layouts can be ordered in a view hierarchy.

UI Rendering

When an activity receives focus, it is requested to draw its view hierarchy, starting from the root node [16]. Drawing the layout has two passes: the measuring pass and the layout pass, both of which are top-down traversals of the view hierarchy. In the measure pass, each view calculates and stores its preferred measurements, with respect to any constraints set by its parent. In the layout pass, each ViewGroup positions its child views using its preferred or imposed measurements, and then all views are drawn to the screen according to their drawing logic.

When an application needs to update its UI, it will invalidate the view that needs to update [6]. The invalidation propagates through the view hierarchy to compute the region of the screen that needs to be redrawn, known as the "dirty region". How the hierarchy is redrawn depends on whether or not drawing is hardware accelerated. The software based drawing model invalidates the view hierarchy and then redraws any view that intersects with the dirty region, whether or not those views have actually changed. The hardware accelerated drawing model records drawing commands in display lists. When a view is invalidated, only the display list of that view is invalidated. The system then draws from the display lists, and if a display list has not been invalidated, it will simply be redrawn without any additional computation. Hardware acceleration was introduced in Android version 3.0 (Honeycomb), and is the default drawing model starting in version 4.0 (Ice Cream Sandwich).

3.2 Performance in Android Applications

Performance on Android is affected by several factors, some of which are software related, and some that are dependent on the specific Android device and its hardware, e.g. CPU (Central Processing Unit), GPU (Graphical Processing Unit), etc., as well as the version of Android the device is running. This section will introduce the main characteristics of the Android platform and Android devices that can cause performance problems, as well as common performance bug patterns.

3.2.1 Performance Characteristics

Below, some of the main characteristics of the Android platform that impact performance are introduced.

Single Threaded System

When an application is launched, the Linux kernel creates a new process for it, with a single thread for execution [20]. By default, application components will run on this single thread, which is the only thread from which the UI can be manipulated,

and where user interaction events can be handled. For this reason, the main thread is often referred to as the UI thread.

Display Refresh Rate and VSync

Each Android device has a certain device and display refresh rate, at which operations and executions are synced to ensure that graphics are rendered correctly and not displayed or updated before they have finished rendering [32]. The standard refresh rate is about 60 frames per seconds, or 16.667 ms per frame. The display sends a signal called VSync every time it is ready to update, at which point the system will search for any changes to the UI that will need updating.

Mobile CPU

The CPUs in mobile devices differ from CPUs in other computers, both in capacity and construction [51]. Although some newer devices can have equally powerful CPUs as some personal computers, mobile CPUs very rarely run on full capacity. Mobile devices have different CPUs used for different tasks: they have low-powered, efficient CPUs that are used for most tasks to save energy, and high performance CPUs that are only used for heavy graphics rendering in e.g. games, or videos. Running the high performance CPU for an extended period of time will cause battery drain, and the device will heat up.

Device Performance

Applications are run as users on the Linux kernel, which share the available resources of the device, including for instance memory and CPU [19]. This means that the performance of an application can impact the performance of the whole device. For instance, an application that uses a lot of memory can cause other applications that are running in the background to be killed in order to free up memory. This can make the entire device appear slow and non-performant to the user when switching applications, because recreating an application takes more time than just restarting or resuming it.

Platform Fragmentation

Android is a very fragmented ecosystem, and there are many devices that are still being sold and used that are running older versions of Android. Figure 3 shows the distribution and usage of Android versions. There have been numerous updates in the newer versions that improve performance, such as the introduction of hardware acceleration (in Android 3.0 Honeycomb), an update of the runtime (in Android 5.0 Lollipop), and the introduction of updating views with VSync (in Android 4.1 Jelly Bean). To ensure that an application can be performant for as many users as possible, developers need to ensure that applications can run smoothly on older devices and older versions of Android as well.

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.2%
4.1.x	Jelly Bean	16	7.8%
4.2.x		17	10.5%
4.3		18	3.0%
4.4	KitKat	19	33.4%
5.0	Lollipop	21	16.4%
5.1		22	19.4%
6.0	Marshmallow	23	4.6%

Figure 3: Android version numbers, code names and distribution rates as of April 2016 [12].

3.2.2 Performance Bug Patterns for Android Applications

As discussed in chapter 2, researchers have found patterns for Android performance bugs, such as heavy operations on the main thread, computation for invisible views, etc. [45]. These are discussed below, along with best practices for improving performance as presented in the Android developer guidelines[3].

Dropped Frames and Frame Rates

As mentioned, Android applications have 16.667 ms to update the UI to meet the 60 fps refresh rate of the device. If a frame takes longer than that to render, that frame is "dropped", meaning the system will not update the UI. This causes the frame rate to drop to 30 fps or less, since the next frame will take 32 ms or more to update. A drop in frame rate is noticeable and jarring for the user.

Blocking the UI Thread

As applications by default have a single thread that is responsible for the UI, any operation that is not responsible for UI updates or handling user input is run on the main thread can be considered a blocking operation. Blocking operations, like the aforementioned examples of network access, bitmap processing, and storage access, could be negligible, but most often they are not. According to Sillars [51], updates between 0-100 ms will be perceived as instantaneous, 100-300 ms might feel a bit

[3] `http://developer.android.com/guide`

slow, 300-1000 ms will cause the user to assume that something is updating in the background, and anything longer than 1000 ms will likely be interpreted as a freeze in the application. On Android, if an application freezes and stops responding to input for more than 5 seconds, an "Application Not Responding" (ANR) dialogue is displayed (Figure 4). To avoid this, all blocking operations should be performed outside of the UI thread.

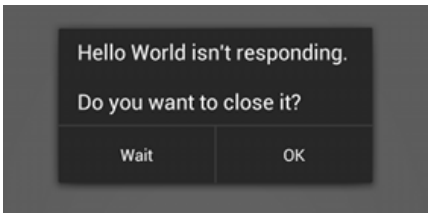


Figure 4: An "Application Not Responding" dialogue, which is shown if the main thread of an application is blocked for more than 5 seconds [17].

Complex View Hierarchies

As evident from section 3.1.1, rendering UI components is a process involving many parts of the system, and UI computations are prone to get very heavy. A complex or poorly implemented UI is likely to lead to poor performance. Updating a deep and complicated view hierarchy will increase the computation needed for the CPU and GPU when updating and rendering display lists, since there is more measuring, layouting, and drawing needed than for a simpler, more shallow view hierarchy [32].

Overdraw

As mentioned, the system updates the UI by drawing each view of the view hierarchy, starting with the root view. The system will draw views on top of each other, ordering children views in front of their parents. This could potentially lead to drawing opaque views on top of other opaque views, effectively drawing the pixels multiple times. This is known as overdraw, and causes unnecessary computation and drawing operations, which can be especially bad for performance during animations [13].

Garbage Collection

The Garbage Collector (GC) removes allocated memory that is no longer being referenced, which clears up memory. GC runs are expensive, especially on older devices. On devices running Android version 5.0 or later (Lollipop), a GC run usually takes about 2-4 ms, and on devices running older versions of Android, a GC run will normally take about 10-20 ms on a fast device [14]. The difference is due to the newer versions using an "ahead-of-time" runtime compiler called ART (Android RunTime), while the older versions uses a "just-in-time" runtime called Dalvik, and these perform GC runs differently. GC runs can be triggered by allocating a lot of temporary objects at the same time, in a loop for instance. They can also

be triggered by memory leaks, which is when unused objects are still referenced somewhere in the system, meaning it will not be deallocated by the GC. Memory leaks mean that there is less memory to use, which will likely increase the rate of GC runs.

3.3 Summary

The Android platform is heavily reliant on the responsiveness of the GUI, which needs to meet a frame rate of 60 fps and an update time of between 0-300 ms to appear smooth and responsive to its users. This means that the UI has about 16 ms to update, which can be difficult as the view rendering process is quite heavy. Any operations that might block the update time, or add complexity to the rendering process in the form of complex view hierarchies or overdraw, will likely cause performance bugs and poor responsiveness.

4 Spotify Debug Tools

During the summer of 2015, the author of this thesis was an Android developer intern at Spotify in Stockholm. During the internship, nine different tools were developed to aid Android designers and developers in their work. This section will introduce those tools, which can be toggled on and off in a menu that slides in on the right hand side of internal versions of the Spotify application, see Figure 5. These tools were evaluated during this thesis work, as explained in section 5.2.

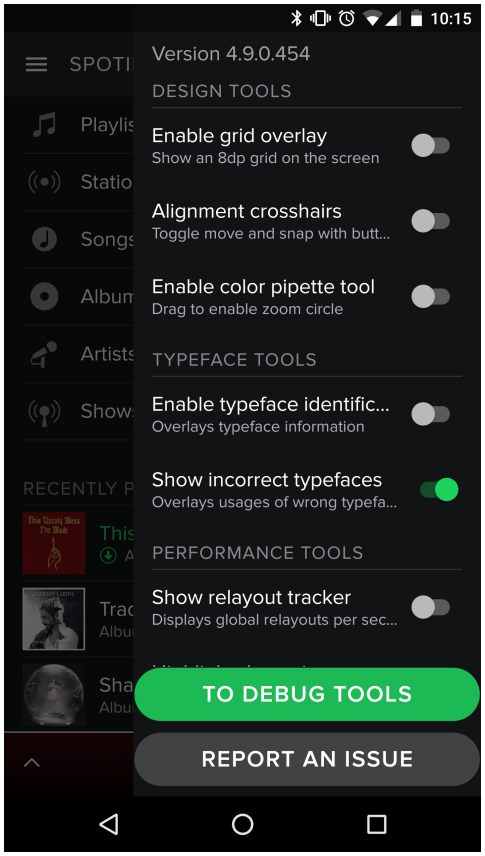


Figure 5: Menu that slides in from the right hand side of the application. Here, all tools are listed along with descriptions, and toggles to enable and disable them.

4.1 Performance Tools

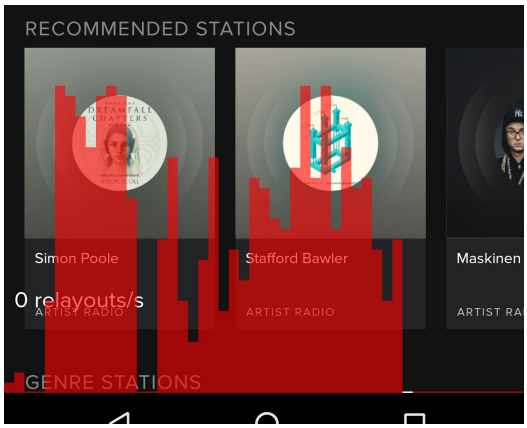
Three of the tools were developed to detect performance bugs in the application. These will be discussed further in the context of this thesis in chapter 7.

Relayout Tracker

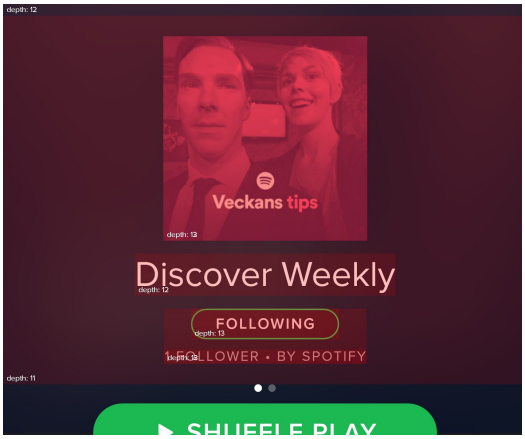
This tool detects global relayouts in the application. A relayout is a callback that is triggered when the view hierarchy goes through a phase of measuring and layout, which can be an expensive operation, as explained in section 3.1.1. The tool displays bars on the screen that indicate how many relayouts are triggered every 0.5 second: the taller the bar, the more relayouts are triggered. The number of relayouts are also stated on the screen. See Figure 6a.

Deep View Highlighter

This tool overlays a translucent red box on all views that are nested deeper than 10 levels in the view hierarchy, as seen in Figure 6b. This makes it easier for developers and testers to find deep and complex view hierarchies, which as previously mentioned might be bad for performance.



(a) Tool that displays the frequency of global relayouts in the application. The taller the red bars are, the more relayouts are triggered.



(b) Tool that overlays views that are deeper than 10 levels in the view hierarchy with a red tint.

Figure 6

View Load Time

This tool measures and displays the time it takes to load a new view after navigation, as seen in Figure 7. This tool already existed before the internship started, it was however added to the debug tool menu and tweaked so it uses the same mechanisms of displaying information as the other tools. It displays the load time text in different colours: green if it is below 300 ms, yellow if it is between 300 ms and 600 ms, and red if it is above 600 ms.

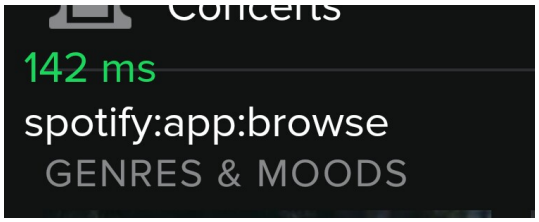


Figure 7: Tool that displays the loading time for the current view, as well as the route name of the view that is currently showing. The green text colour indicates that the view load time is below 300 ms, which is desirable.

4.2 Design Tools

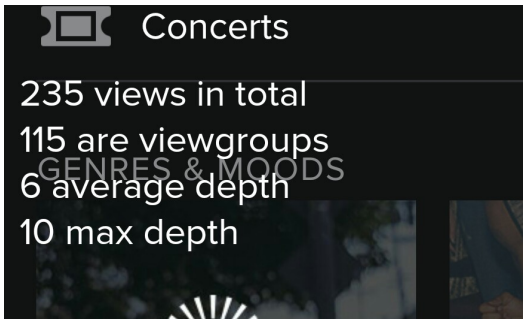
The following tools were designed to make it easier for designers and QA (quality assurance) to discover design related bugs in the application. Although these tools are not of further interest in the context of this thesis, they are introduced to give context to the tool evaluation explained in section 5.2.

View Statistics

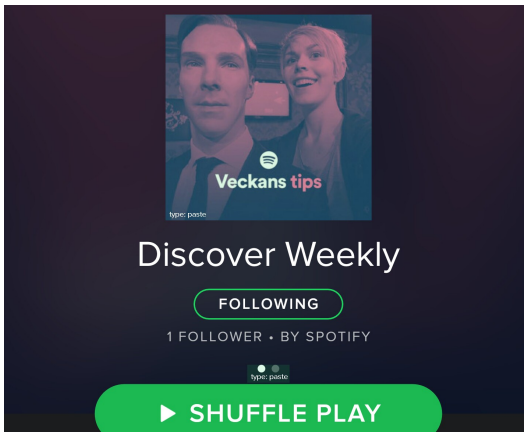
This tool shows some descriptive statistics about the views currently on the screen. It displays the total number of views currently on screen (visible and otherwise), how many of those views that are ViewGroups, the average view depth, and the maximum view depth. See Figure 8a.

Component Identification

This tool overlays a translucent blue box on any component that is a custom component, rather than one of the system components that the Android toolkit provides, as seen in Figure 8b.



(a) Descriptive statistics about the views currently on screen, such as the number of views and viewgroups, the average view depth, etc.



(b) Tool that identifies custom components in the application. Here, the image and the dots are identified as custom views.

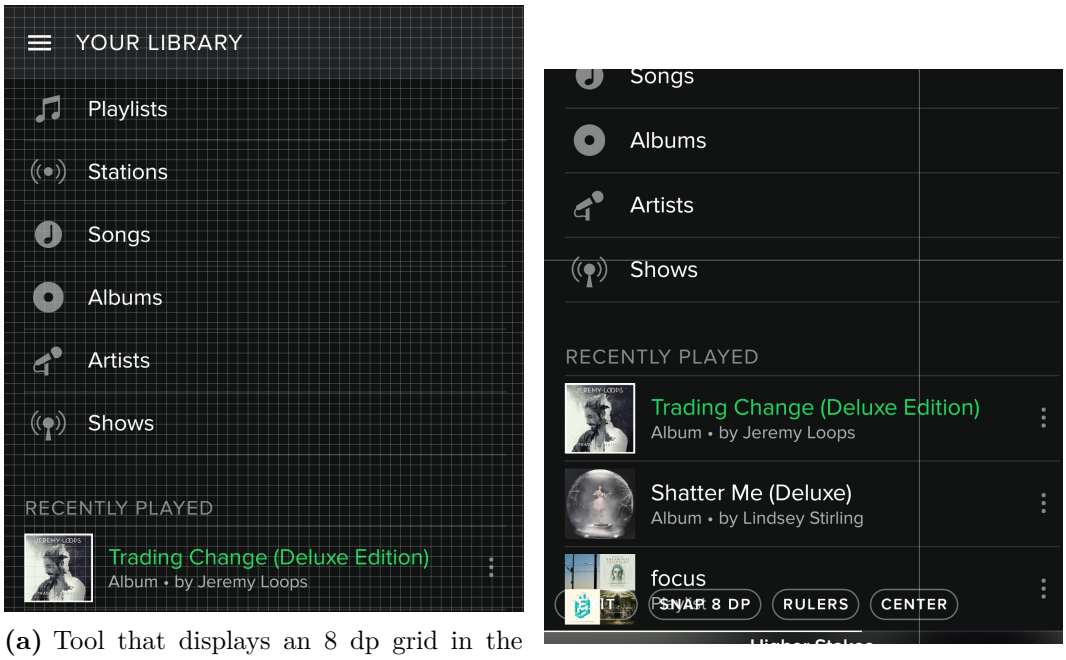
Figure 8

Grid Overlay

Android design guidelines are based on the so called 8 dp grid, where dp, or dip, stands for density independent pixel. This unit is used so that views are displayed the same on low or high density display. This tool overlays the application with an 8 dp grid to simplify the detection of whether or not views are properly aligned to each other and to the grid. See Figure 9a.

Alignment Crosshairs

This tool overlays crosshairs consisting of two lines on the screen, one horizontal and one vertical, which can be moved across the screen to check alignment of views. The crosshairs can be moved freely, or be set to snap to an 8 dp grid or a 4 dp grid. See Figure 9b.



(a) Tool that displays an 8 dp grid in the application, to make it easier for developers and designers to ensure that the UI is properly aligned.

(b) Tool that displays alignment crosshairs for verifying view alignment in the application.

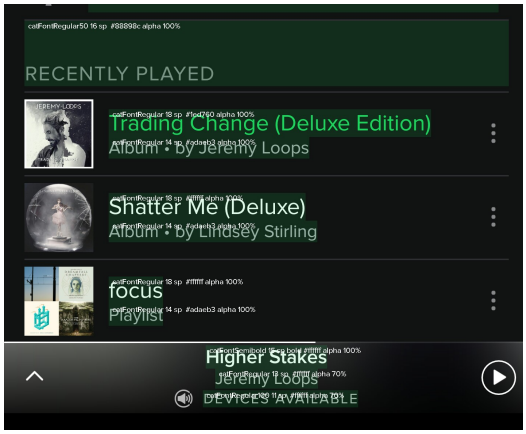
Figure 9

Typeface Identification

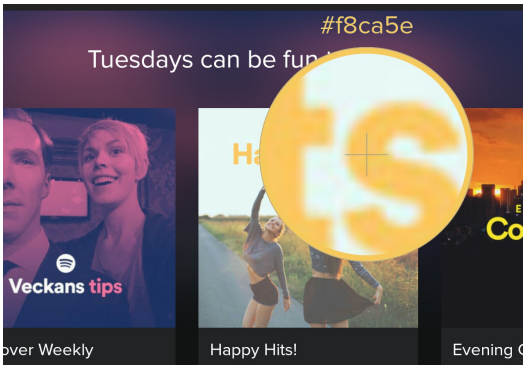
This tool overlays TextViews, i.e. text elements, with a translucent green box, along with a description of the font it uses. It displays the font family, the font size, the HEX value of the text colour, the alpha value (transparency value) of the text, and whether or not it has a set style, such as bold or italic. See Figure 10a.

Colour Pipette Tool

This tool displays a circle that can be moved across the screen, and that acts as a magnifying glass: the circle displays the area beneath it with a 10x magnification. It also extracts the colour code of the pixel in the middle of the magnification, which is marked with small crosshairs for the user, as shown in Figure 10b. The tool displays the HEX value of that colour, and if it matches any of the defined theme colours, the name of that colour resource is shown as well.



(a) Tool that displays information about typeface, such as the font family used, the font size, etc., for text views by overlaying it with a green tint.



(b) Colour pipette tool that can be used to extract colour information from the UI. The circle acts as a magnifying glass, and extracts information from pixel in the middle of the circle.

Figure 10

Incorrect Typeface Detection

This tool works similarly to the typeface identification tool described above, but will only highlight text elements that does not use the correct typeface as specified by Spotify's design guidelines. The overlays of the tool are similar to those in Figure 10a, except that they include the text "Incorrect Typeface".

5 Method

This chapter presents the work methodology used during this thesis. The areas described below include a two-tiered literary study, an evaluation of existing tools, a case study conducted on the Spotify application, and the development of a new tool. The literary study has been conducted throughout the thesis work, and the other tasks have been carried out in parallel during different periods.

5.1 Literary Study

To understand the context of this thesis work, an extensive literary study was carried out. Initially, the study focused on related research about software performance and topics related to that, including mobile development, testing practices, practices for fixing bugs, etc. The reviewed research mainly consisted of articles and conference papers found through Google Scholar[1]. Some of the keywords and phrases used to find the research included "Android performance", "buggy interfaces", "debug interfaces", "measure performance", "performance bugs", etc. Papers and articles reviewed were also found through the reference lists of other papers and articles. The reviewed material and the results gained from this phase of the literary study are summarised in chapter 2.

A separate facet of the literary study included research about Android, in order to understand how the platform works, mainly regarding performance. Material and information consumed for this work was predominately produced by Google, mainly on their Android developer website[2], through recordings from conference talks given by Google employees, or through videos on the Google Developer YouTube channel[3]. The book "High Performance Android Applications" [51] was also an important source for this research. A summary of the gained insights is presented in chapter 3.

5.2 Evaluation of Existing Tools

To understand how the tools presented in chapter 4 have been performing after the author's internship ended, whether they have been effective in detecting bugs and verifying bug fixes, and to get comparative data to evaluate, a survey was created. The survey consisted of 11 questions:

[1] `https://scholar.google.com/`
[2] `http://developer.android.com`
[3] `https://youtube.com/GoogleDevelopers`

- One demographic question.
- Five close-ended questions regarding knowledge and usage of the tools.
- Three open-ended questions about possible improvements, additional tools, or comments about the tools.
- One open-ended motivation.
- One open-ended question about whether the respondent would like to be of further help to the thesis.

The survey was created using Google Forms, and was sent out via email and internal communication channels to Android developers, testers and designers across Spotify. The survey along with an introductory text is listed in Appendix A.

5.3 Case Study with the Spotify Application

In order to apply the findings from the literary study, and to get practical knowledge of different tools that exist for measuring and improving performance, a case study was carried out. The case study included measuring and profiling the Spotify application using different tools, and exploring how the Spotify application is built with regards to common performance bug patterns and practices. The main goal was to get a sense of the performance profile of the Spotify application, and identify areas where the performance could be improved.

Profiling Device

To get consistent measurements, and to eliminate as many contextual factors (such as other applications using resources in the background, device storage differences, different technical specifications, etc.) as possible while measuring, a single device was used. Some requirements were set on the profile device in order to get measurements as representative of an average Spotify user as possible, including that the device could realistically run the Spotify application smoothly, but not be performant enough to disguise any eventual performance bugs. The device used for the profile was a Samsung Galaxy S3 (GT-I9305) running Android version 4.3 (Jelly-bean). It was chosen with the help of a QA (quality assurance) at Spotify as the best device available to fit the requirements mentioned. The S3 was released as a high-end device in 2012 with a quad-core 1.4GHz CPU. The device had 16 GB of storage, no external SD card, and was connected to WiFi during the profiling. The Internet connection had an upload and download speed of about 40 MBit/s, and the device was connected to a computer at all times, both to enable debugging and monitoring using different desktop tools, as well keeping any battery inconsistencies to a minimum.

GPU Overdraw Profiler

One of the on-device developer tools[4] that the Android platform provides is an overdraw detection tool, which overlays the UI with different colours based on the amount of overdraw [13], as illustrated by Figure 11. Blue tint means 1x overdraw, or that the pixels have been drawn twice, green tint means 2x overdraw, lighter red tint means 3x, and deeper red is 4x overdraw, meaning the pixels have been drawn 5 times or more. For measuring overdraw in the Spotify application, this tool was enabled during exploration of the UI.

Figure 11: The tint colours used in the "Debug GPU Overdraw" tool that the Android platform provides. Blue tint means 1x overdraw, or that the pixels tinted with blue have been drawn twice, green that they are drawn three times, etc. [13].

GPU Usage Profiler

The GPU usage profiler is another tool that the Android platform provides, and can be run on-device or from Android Studio [21]. The tool displays stacked bars on the screen or in a graph monitor in Android Studio, and shows real-time GPU usage for UI updates. The stacked bars show how much time is spent on different operations, such as measuring, layout, drawing, etc. The tool also displays a green, horizontal line that represents the 16 ms frame update limit, which makes it easy to detect any dropped frames. Profiling the application with this tool was done by enabling the tool and exploring the UI, to detect any areas where the GPU usage would exceed the 16 ms limit, causing the bars to exceed the line, and also result in stutters in animations.

[4]These tools are available in the hidden "Developer Options" in the device settings. These options are displayed if the "Build number" field in the "About phone" section is tapped 7 times.

Profiling Application Startup Time

To measure the start up time of the application, the view load time tool described in section 4.1 was used. The results from this tool are not fully representational for the actual startup time of the application, as the tool itself is not initialised at the very beginning of the startup process, but it still provides descriptive and comparable data.

AndroidDevMetrics

AndroidDevMetrics [53] is an open source tool that was initially released on GitHub in March of 2016. It measures activity lifecycle metrics and dropped frames, as well as the frame rate during the frame drop. The lifecycle metrics includes the average time for how long the activity takes to perform lifecycle callbacks like onCreate, onStart, and onResume, as well as the time taken to perform a layout of the activity UI. This tool was added to the Spotify application and was run during exploration of the UI.

HierarchyViewer

HierarchyViewer is a tool that can be run as a standalone desktop application, or as part of the Android Studio Device Monitor [15]. An overview of the tool is shown in Figure 12. The tool displays each view in the view hierarchy as a box, and the boxes are connected in a tree like structure. The user can choose to profile the view hierarchy, or subsets of it, to evaluate the time it takes to measure, layout, and draw the view hierarchy. The results from the profiling are visualised on the boxes with three dots, each corresponding to the view rendering phases. A green dot indicates that it was faster than at least half of the other views, a yellow dot that it was faster than the bottom half of the other views, and red that it is in the slowest half of the views. The result can also be viewed as explicit load times by inspecting the boxes. The HierarchyViewer tool was used to inspect and profile different views in the Spotify application.

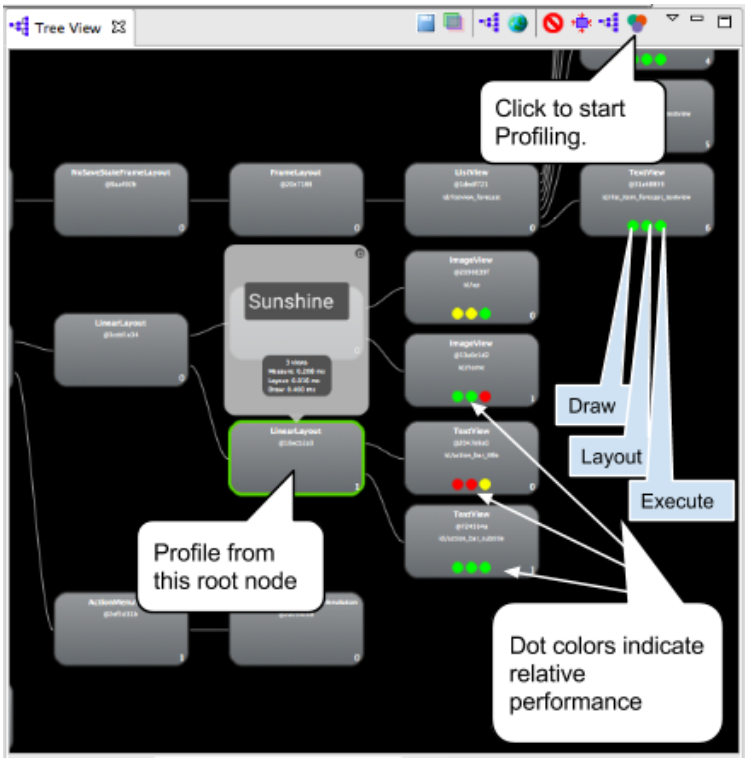


Figure 12: A screenshot of the HierarchyViewer tool, where each of the boxes represents a view in an activity. The measure, layout, and draw sequences of the views can be profiled in the tool, and the results are visualised as dots on the boxes [15].

Best Practice Evaluation of the Spotify Application

To explore how the Spotify application is architectured and implemented with regards to known performance bug patterns and best practices (as discussed in chapter 3), an investigation was carried out. The code base was inspected, as well as tools and practices used for developing and testing the application, with the guidance of some Android developers at Spotify.

5.4 Development of a New Tool

During the thesis, a new tool for simplifying the removal of overdraw was developed. The idea for the tool emerged from measuring overdraw during the case study. It was integrated as a part of the Spotify application, like the tools described in chapter 4. The tool was developed with Java using the Android framework, using Git[5] as a version control system. A version control system makes is possible for multiple developers to work on the same code base, and Git is the version control system that developers at Spotify use to manage their code bases.

[5] `https://git-scm.com`

6 Results

This section presents the results obtained during the thesis work, which includes the results from the evaluation survey, results from the case study with the Android application, and a presentation of the new tool.

6.1 Results from Tool Evaluation

The tool evaluation survey had in total 19 respondents (out of about 100 active contributors to the Android code base at Spotify, as well as designers and QAs (quality assurance)). The results of the close-ended questions are displayed in Table 1.

Question	**Results**
What's your occupation at Spotify?	16 Android developers and managers, 2 designers, and 1 QA.
Were you aware of the existence of the debug tool drawer?	19 yes.
Have you used any of the tools in the drawer?	14 yes, rest no or N/A.
If yes, which tool has been most helpful/useful?	3 relayout tool, 2 grid tool, 2 deep view highlighter, 2 incorrect typeface tool, 1 view statistics, 1 colour pipette tool, 1 view load time tool, 1 alignment crosshairs.
What do you use the tools for?	12 development purposes, 2 design purposes, 1 QA/testing purposes.
How often would you say you use the tools or the drawer?	7 weekly, 4 monthly, 3 every few month, 3 "never really...".

Table 1: The questions and answers from the close-ended questions in the tool evaluation survey presented in section 5.2.

Motivations for choosing the most useful tool included that the relayout tool had proved useful for finding performance issues, the deep view highlighter had discovered unnecessary depth, the view load time tool ensured load time quality, the grid tool was useful for checking sizes, the incorrect typeface tool for discovering errors in typeface component usage, and the pipette tool since that was the only tool that particular respondent had used.

As for suggestions for tool improvements and additions, there were wishes for more

sophisticated component identification, a tool to show what the application is doing if it is frozen, and methods for comparing performance between versions and code changes.

There were also some thoughts about communication with regards to the tools: when to use them, how to use them, how to make them difficult to ignore or avoid, etc. As one respondent put it:

> "The approach with having tools and visualisations within the app is great but also easy to miss/ignore. Doing some checks and stats on new additions during the build and test phase will probably be very useful [if and only if] they can be stable enough to be trusted."

Another respondent answered:

> "The drawer seems to consist of many useful tools now that I actually looked through them. To me personally, the issue is really to know when to use what tool."

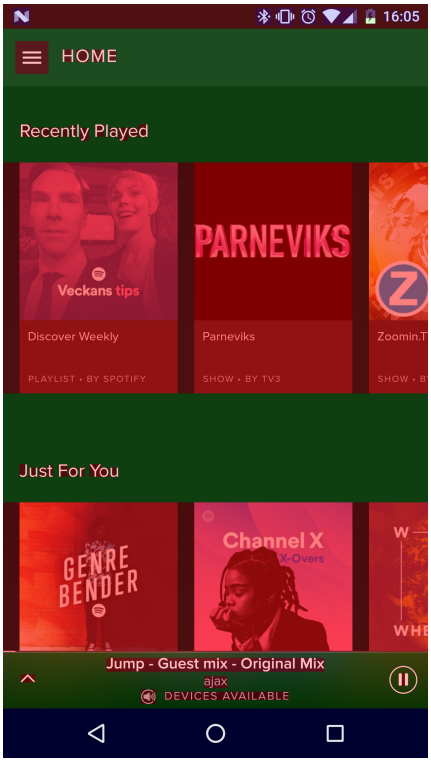
Some of the tools, mainly the relayout tool, the incorrect typeface tool, and the deep view highlighter, have been used to detect and verify fixes for bugs since they were introduced. As an example, the relayout tool used to peak at 60 relayouts per second during some interactions, e.g. scrolling in certain lists. Now, it rarely exceeds 2-4 relayouts per second. Likewise, the deepest view hierarchies were around 16 or 17 levels deep, and are now around 11 or 12 levels deep.

6.2 Results from the Case Study

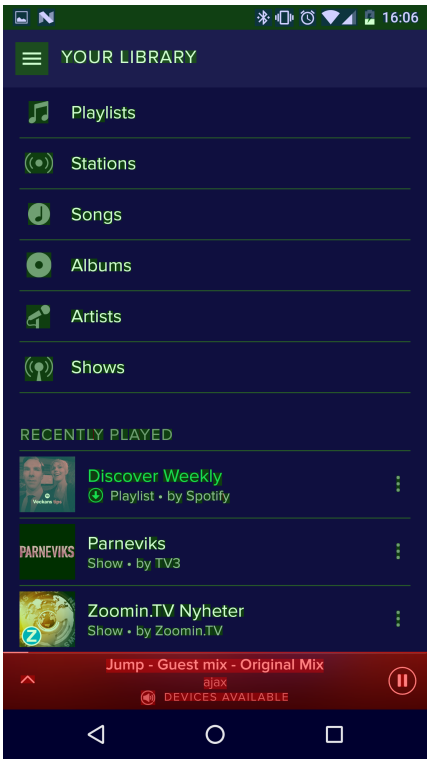
The performance profile was gathered over a few weeks from a Samsung Galaxy S3 as described in section 5.3.

GPU Overdraw

Results from the GPU Overdraw tool showed that the Spotify application was suffering from overdraw throughout the application. Figures 13a and 13b shows screenshots from the Home view and the Library view with the overdraw tool enabled, where Home (Figure 13a) has 2x and 4x overdraw, as visualised by the green and red tint, and Library (Figure 13b) has between 1x and 4x overdraw. Other explored parts of the application were also suffering from at least 1x overdraw, with the exception of one view that displays queued music.



(a) Overdraw for the Home view with 2x (green tint) to 4x (red tint) overdraw.



(b) Overdraw for the Library view with between 1x (blue tint), 2x (green tint), and 4x (red tint) overdraw

Figure 13

GPU Usage Profiler

The GPU usage profiler was used throughout the application, and testing included rigorous user interactions, e.g. scrolling through lists very fast, and performing extensive dragging operations, in order to detect as many problematic areas as possible. Views in the application where the GPU profiler repeatedly exceeded the 16 ms rendering limit, resulting in dropped frames and stutters in animations, were:

- Home view - While scrolling.
- Running category - While scrolling.
- List views, e.g. playlists, artists - When scrolling to the top of the list.
- Now Playing View/Player - When skipping songs, and when track titles were looping.
- Library view - While scrolling.
- Party tuner view - While loading and updating.

In general, it appeared as if updating placeholders to loaded images was the main cause of the exceeding rendering times. Upon an informal inspection of logcat, a tool in Android Studio that displays logs of system messages, it was found that loading images for thumbnails etc. caused frequent GC runs.

Application Startup Times

In total, there were 129 measurements, whereof 110 were measured from backing out of the application, and then resuming it again, and 19 were measured after the application was force stopped via the Android settings menu, ensuring a "cold start" (complete reset of resources and processes) of the application. The results are presented in Table 2.

	Resume/restart	**Cold start**
Mean load time	687.74 ms	1608.53 ms
Standard deviation	470.90 ms	149.51 ms
Median load time	526 ms	1587 ms

Table 2: The mean load time, the standard deviation, and the median load time for the startup time measurements.

AndroidDevMetrics Results

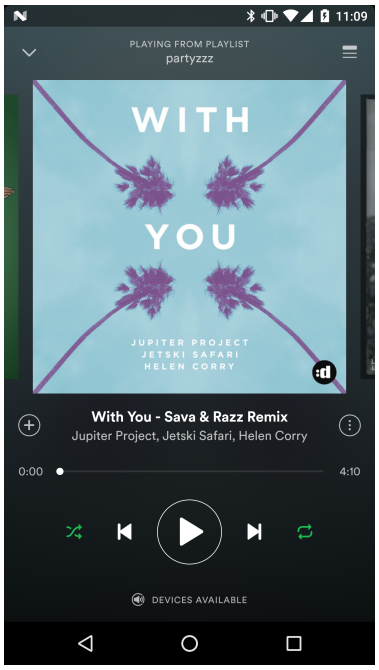
The results from using the AndroidDevMetrics tool can be seen in Table 3. As mentioned, the results from the AndroidDevMetrics tool are averaged based on the number of activity instances, and these results are further averaged over the number of measurements, denoted as tries.

Activity	**Tries**	**onCreate (ms)**	**onStart (ms)**	**onResume (ms)**	**Layout (ms)**	**Instances**	**Dropped frames**
MainActivity	11	135.82	77.58	39.33	555	79	727
PlayerActivity	6	11	19.29	30.43	156.17	29	379
EditablePlayQueueActivity	5	112.60	12.40	26	164.4	7	28
LoginActivity	1	28	18	20	337	1	1
DeviceActivity	2	9.50	49	27	144	2	21
ManualTempoActivity	1	50	6	20	134	1	1

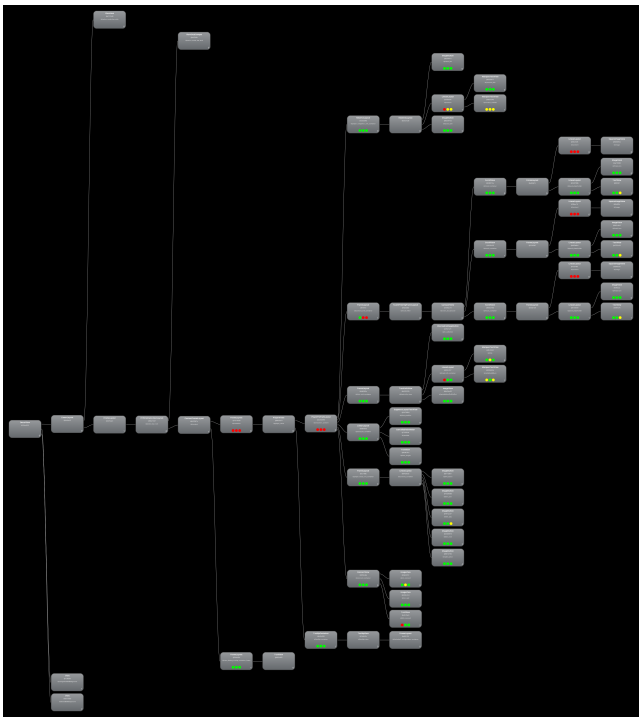
Table 3: The results from using the AndroidDevMetrics tool in the Spotify application.

Results from HierarchyViewer

One result from using the HierarchyViewer can be seen in Figure 14. The inspected view hierarchy is from the PlayerActivity, or "Now Playing view", shown in Figure 14a, and Figure 14b displays the view tree in HierarchyViewer. Worth noting is that the first 4 nodes are rendered from the system rather than the Spotify application.



(a) A screenshot from the PlayerActivity in the Spotify application.



(b) A screenshot from HierarchyViewer of the PlayerActivity view hierarchy.

Figure 14

Results from Best Practice Evaluation

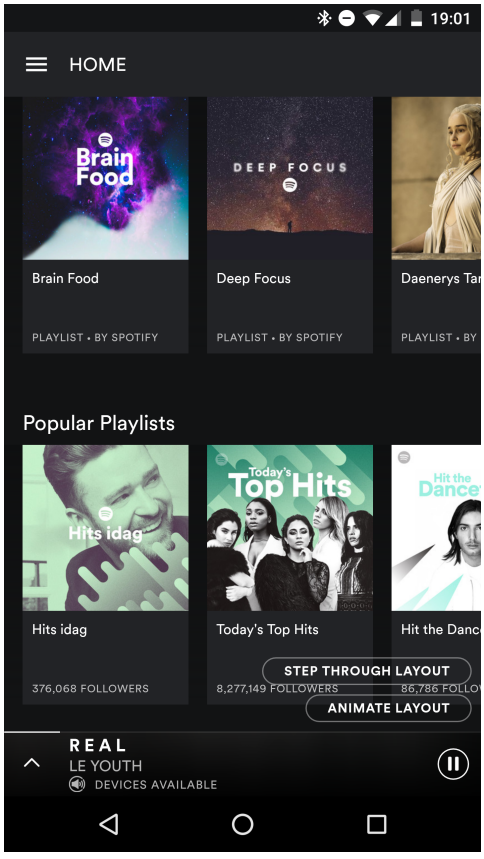
There are some measures taken against the common bug patterns discussed in chapters 2 and 3. There are several testing tools used during development at Spotify, including static analysis tools (Lint[1], FindBugs[2], and CheckStyle[3]). These perform checks for performance bug patterns, e.g. non-UI operations on the main thread, and warnings about opaque views that might introduce overdraw. Other tools, such as LeakCanary[4], are used for detecting memory leaks.

To ensure application performance across the platform fragmentation, QAs at Spotify have an extensive suite of testing devices, including older devices, lower end devices, and devices running older versions of Android. Best practices regarding UI performance, such as deep and complex view hierarchies, and overdraw avoidance, have historically not been considered as much, although this thesis work as well as prior work by the author are measures to improve that.

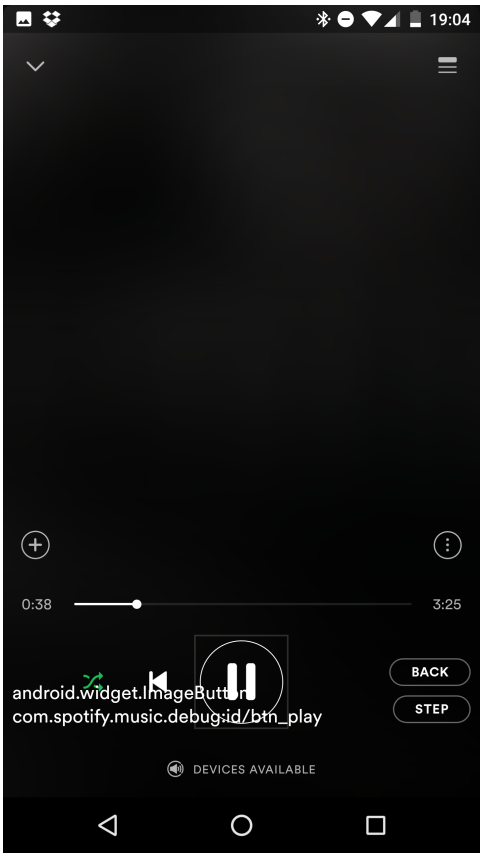
[1] `https://developer.android.com/studio/write/lint.html`
[2] `http://findbugs.sourceforge.net`
[3] `http://checkstyle.sourceforge.net`
[4] `https://github.com/square/leakcanary`

6.3 View Hierarchy Traversing Tool

This tool was developed to simplify the process of removing overdraw in an application, as a result of the results from the overdraw profiling. The main functionality of the tool is to acquire the current view hierarchy of the application, hide all visible views, and stepping through them view by view, displaying them again. When a hidden view is displayed, the tool highlights the view with a white border, displays the component name, and the view ID if there is one. When the tool is enabled, there are two buttons in the bottom right corner of the application, as shown in Figure 15a. The "Step Through Layout" button allows the user to step through the layout view by view with two other buttons, both backwards (hiding views again) and forwards. The "Animate Layout" button will animate the layout, displaying a new view every 250 ms. While animating, the user can choose to pause and resume the animation to inspect any single view.



(a) A screenshot of the Home view with the new tool enabled. The buttons in the bottom right corner let the user either step through a layout view by view, or animate it.



(b) A screenshot from the Spotify application while the step-through functionality of the new tool is enabled. Here, the play button has just been displayed, indicated by the square white border.

Figure 15

With the Android overdraw detection tool enabled, the tool can be used to identify which view is causing the overdraw, as the overlay tool will tint the views as they

are shown while stepping through the hierarchy. The traversing tool was used to remove overdraw throughout the Spotify application when it had been developed, as it made it easier to detect which views and layouts introduced the overdraw. It was found that the majority of the existing overdraw was introduced by fragments that had opaque backgrounds in the same colour as the window background of the application, and when these fragments were set to have transparent background instead, almost all of the detected overdraw was removed.

7 Discussion

The following sections include discussions about the results of this thesis work, answers to the research questions asked in the introduction of the report, any limitations, problems, and uncertainties in thesis work, proposed guidelines for improving performance in Android applications, as well as ideas for future work in this area.

7.1 Results and Findings

When the previously developed tools were introduced in the summer of 2015, they received praise from other developers at Spotify. Especially the performance tools, e.g. the relayout tool, and the deep view highlighter, detected and visualised bugs that were completely unknown to the developers at the time. The results from the tool evaluation supported the fact that these tools have shown potential, and that the seemingly simple action of visualising something by tinting or flashing it is an effective approach to finding performance bugs. The performance tools were amongst the most popular tools, and suggestions given in the evaluation have been useful for gathering ideas to explore during the thesis work.

The overall goal of this thesis was to research mobile performance, and use practices and strategies to understand how developers can and should work with regards to performance. The case study that was carried out proved valuable in meeting that goal. The results gained from the study, for instance the data on startup times, and from AndroidDevMetrics, are interesting indicators for areas where the Spotify application still has work to do. Some of the activities measured with AndroidDevMetrics instantiated above the recommended 0-200 ms range for appearing instantaneous to the user, especially MainActivity. Such is also the case for the recorded startup times. Even though the Spotify application fares well with regards to best practices, such as avoiding blocking operations on the main thread, and managing memory to avoid GC runs, it still faces challenges with other aspects of performance, for instance working well across the platform fragmentation.

By profiling and measuring performance data, one acquires results which can then be acted upon accordingly. For instance, during the thesis, the findings about overdraw led to the development of a tool to find the sources of the overdraw, and ultimately in fixing and removing most of it. Likewise, the usage of the GPU profiler tool has uncovered that there are multiple GC runs when updating placeholders to loaded images while scrolling, which can be an action point for further work. This thesis only scratched the surface on performing an extensive performance profile of an Android application, but has nonetheless identified problematic areas, and attempted to take action to remedy them.

7.2 Answering the Research Questions

In order to focus the scope of the thesis, the following research questions were formulated.

- What are performance bugs, and how do they impact mobile development and platforms?
- What causes performance bugs in Android applications?
- Can causes of performance bugs be detected visually? Can visualisation be a valid strategy for performance testing and tooling?

The first two research questions were the basis of the literary study. The resulting answers to these questions are presented in chapter 2 and 3 respectively. In summery, performance bugs are defined by causing slow execution times rather than deterministic errors, and are affected by many aspects of the mobile context, such as limited hardware capabilities, and of the Android operating system as well. The third research question is more interesting, and it was initially derived from the results of the tools developed prior to this thesis. The view hierarchy traversal tool presented in section 6.3, along with some of the previously developed tools presented in chapter 4, have shown that visualisation can be used to both find and understand new performance bugs, as well as show new ways of fixing and validating them. For the Android platform, there is much going on underneath the hood, and with visualisation tools, it could be possible to get a deeper understanding for how things really work. As found, understanding common causes for performance bugs will not necessarily be helpful for performance debugging, if there is no feasible way of verifying them.

This thesis has focused mainly on performance bugs introduced by UI components. Using visualisation to detect UI bugs is quite natural, seeing as it is already the only visual part of the application. In that sense, detecting hidden views or deep view hierarchies is fairly straight forward. Another interesting approach is to attempt to visualise non-UI parts of the application, like callbacks, triggers, broadcasts, etc. This approach is used in the relayout tool for instance, which counts and visualises relayout callbacks in the application. Examples of other candidates for such visualisation are garbage collection events, and blocking operations. Visualisation is definitely a valid strategy for detecting sources for performance bugs, as demonstrated by the tools discussed in this thesis, and there are more areas still that might benefit from such a strategy.

7.3 Limitations

Spotify implements continuous delivery, which means that code is developed and deployed continuously. The Spotify application is updated every other week, and there are always new changes to the code base. In the context of this thesis, this means that there is no real stable testing environment, and changes that might improve performance could be reset by changes that lower the performance in the

next update. Performance testing is difficult enough without the added error source of constant change. The results from the case study might not be comparable or representative of the application as it is today, and it could potentially have been skewed as it was measured as well. Even if the code for the Spotify application in itself was not updated, changes to the backend might still impact performance. This is one reason as to why the thesis has focused more on UI related performance, as the UI logic is one of the more stable parts of the code base, and is not subjected to frequent change.

7.4 Proposed Guidelines

This section proposes some guidelines based on the results of this thesis work, and the discussion above. Many of them echo guidelines set forth by the Android team as well as by researchers in the field.

7.4.1 Go Looking for Problems

Where there is code, there are bugs, including performance bugs. Given that there are bugs, in order to fix them, one first needs to find them, so go looking. There are tools, best practices, and bug patterns for performance bugs which are great places from which to start, and using tools to profile and measure application performance will produce data that can be used as benchmarks for comparison later on.

7.4.2 Understand What It All Means

When a problem is located, it needs to be understood. Not knowing what a problem means or entails will make it impossible to solve it. Use tools to gain knowledge about the problem, visualise if possible, and again, looking to best practices might be a good place to start.

7.4.3 Do Your Best To Fix Things

Understanding the problem will hopefully also give some insights as to how the problem can be solved. In that case, do it. In other cases, you might need to try new things. If your toolbox does not include the tools needed to fix the problem given what you know about it, maybe introducing a new tool could help. Try to understand the problem with the help of the new tool, and iterate if necessary.

7.4.4 Make Sure It Works

When you think you have solved the problem, prove it. Compare the results to the previous results, and verify your solution. Data very rarely lies, but as this is about performance, testing it out for yourself and getting a sense of whether or not it feels better is also a valid approach. Objective data is however always better.

7.4.5 Repeat And Get Involved

This is a cycle that never ends, as there will always be new code, and there will always be new bugs. With a focus on performance bugs, it is a good approach to involve others in the process, as it is futile to patch performance bugs if another developer is busy adding new ones. Tools are again a good approach here, as they can make it easier for everyone to find and verify changes. So take what you have learned, and go looking again.

7.5 Future Work

This section discusses ideas that could not be explored properly during the thesis, but would be interesting to explore in further work in this area.

Other Tools

There are other tools provided by the Android platform that can be helpful in locating performance bugs, such as Systrace, Traceview, and StrictMode. These tools were outside the scope of this thesis, but this section will briefly introduce them as tools of interest. Systrace is a tool that records a trace of execution times for all processes and CPUs [5]. Inspecting the trace can show in detail what causes frames to be dropped, or what operations might block other updates. Traceview is a tool that gathers information about method calls, and can be used to locate heavy operations [22]. StrictMode is a tool that detects violations of different predefined policies, and reports these either via logs, or via an on-device tool that flashes the screen [24]. Common policies are set up around slow and blocking operations, memory leaks, etc., and the usage of visualisation in this tool makes it interesting in the context of this thesis.

Ideas for Visualisation

As this thesis and the tools developed before and during the thesis focus mainly on visualisation, some ideas for other visualisation tools that could detect performance bugs are:

- Visualising empty viewgroups and views that for some reason do not affect the final GUI experience.
- Visualising frequent measuring and redrawing of views.
- Visualise warnings about usage of RelativeLayouts in the root of the view hierarchy, or LinearLayouts with weights. These layouts will measure their children at least twice in order to position them correctly, which means that a measuring phase with a RelativeLayout as a root view will increase execution times for the measuring phase exponentially.
- Visualise GC runs by flashing the screen.

- Implementing an FPS counter, in order to visually track frame rate drops on screen.
- TTI-meter (Time To Interaction), to ensure that pauses are kept to a minimum, and maintaining a responsive experience for the user.

Performance Profiling Model

An idea that was considered during the thesis work was to visualise the performance profile in a polar chart, as in Figure 16. Each pie slice could represent one performance aspect, such as memory usage, view hierarchy complexity, overdraw, etc., each with its own scale of values that can be used to grade the performance. This model could either be manually filled out by hand, but it could also potentially work as an automated integration tool, which can show and test the application in a much more deterministic fashion than with manual measurement. In that case, it could provide a performance score based on the statistics derived from the mean or median of the obtained values.

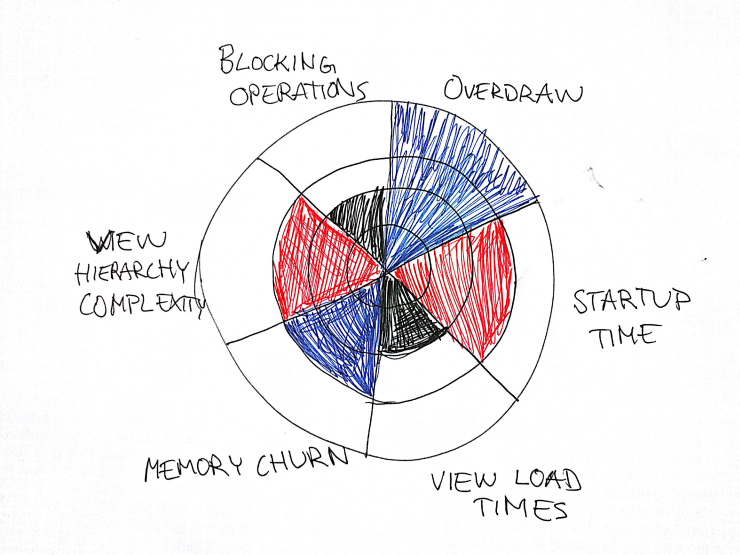


Figure 16: An concept for a tool which uses a polar chart to visualise a performance profile. Each pie slice in the polar chart represents an aspect of performance.

7.6 Conclusions

This thesis has explored performance of Android applications, and the numerous aspects that impact it. As found in chapter 2, the research area of software performance is an interesting and growing field of study, especially for software performance on mobile platforms as they also face challenges with regards to the mobile context. The Android platform in particular has many characteristics that affect performance, as presented in chapter 3, but researchers and developers alike are

working to develop tools, best practices, and strategies to help themselves and others to improve performance across Android applications and the platform itself.

Some of these tools and practices have been used and evaluated during the case study with the Spotify application, along with the tools developed prior to this thesis. Key findings of this thesis are that although improving performance can be a difficult and tedious process, there are plenty of tools, best practices, guidelines, and resources for developers to use in the process. This thesis also highlights promising results for using visualisation for detecting sources for performance problems, which can simplify the process of detecting that which cannot be seen. This approach could make it easier for developers to understand performance problems, as well as the architecture of their own applications.

References

[1] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application texting. In *ICSTW'11*, pages 252–261, 2011.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato. A toolset for gui testing of android applications. In *ICSM'12*, pages 650–653, 2012.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *ASE'12*, pages 258–261, 2012.

[4] Android. Activities, 2016. `http://developer.android.com/guide/components/activities.html`, accessed 2016-04-12.

[5] Android. Analyzing ui performance with systrace, 2016. `https://developer.android.com/studio/profile/systrace.html`, accessed 2016-05-24.

[6] Android. Android drawing models, 2016. `http://developer.android.com/guide/topics/graphics/hardware-accel.html#model`, accessed 2016-04-13.

[7] Android. Android interfaces and architecture, 2016. `https://source.android.com/devices/`, accessed 2016-04-13.

[8] Android. The android source code, 2016. `https://source.android.com/source/index.html`, accessed 2016-05-02.

[9] Android. Android, the world's most popular mobile platform, 2016. `http://developer.android.com/about/android.html`, accessed 2016-02-25.

[10] Android. Broadcast receivers, 2016. `http://developer.android.com/reference/android/content/BroadcastReceiver.html`, accessed 2016-04-12.

[11] Android. Content providers, 2016. `http://developer.android.com/guide/topics/providers/content-providers.html`, accessed 2016-04-12.

[12] Android. Dashboards, 2016. `http://developer.android.com/intl/es/about/dashboards/index.html`, accessed 2016-05-02.

[13] Android. Debug gpu overdraw walkthrough, 2016. `http://developer.android.com/tools/performance/debug-gpu-overdraw/index.html`, accessed 2016-05-09.

[14] Android. Debugging art garbage collection, 2016. https://source.android.com/devices/tech/dalvik/gc-debug.html, accessed 2016-04-13.

[15] Android. Hierarchy viewer walkthrough, 2016. https://developer.android.com/studio/profile/hierarchy-viewer-results-walkthru.html, accessed on 2016-05-24.

[16] Android. How android draws views, 2016. http://developer.android.com/guide/topics/ui/how-android-draws.html, accessed 2016-04-13.

[17] Android. Keeping your app responsive, 2016. http://developer.android.com/training/articles/perf-anr.html, accessed 2016-02-29.

[18] Android. Layouts, 2016. https://developer.android.com/guide/topics/ui/declaring-layout.html, accessed 2016-04-13.

[19] Android. Managing your app's memory, 2016. http://developer.android.com/training/articles/memory.html, accessed 2016-04-13.

[20] Android. Processes and threads, 2016. http://developer.android.com/guide/components/processes-and-threads.html, accessed 2016-05-09.

[21] Android. Profiling gpu rendering walkthrough, 2016. http://developer.android.com/tools/performance/profile-gpu-rendering/index.html, accessed on 2016-05-10.

[22] Android. Profiling with traceview and dmtracedump, 2016. https://developer.android.com/studio/profile/traceview.html, accessed 2016-05-24.

[23] Android. Services, 2016. http://developer.android.com/guide/components/services.html, accessed 2016-04-12.

[24] Android. Strictmode, 2016. https://developer.android.com/reference/android/os/StrictMode.html, accessed 2016-05-24.

[25] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, Martin Schäf, Ishan Banerjee, and Atif M. Memon. Lightweight static analysis for gui testing. In *ISSRE'12*, pages 301–310, 2012.

[26] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Ray. In situ understanding of performance bottlenecks through visually augmented code. In *ICPC'13*, pages 63–72, 2013.

[27] Stefanie Beyer and Martin Pinzger. A manual categorization of android app development issues on stack overflow. In *ICSME'14*, pages 531–535, 2014.

[28] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. An empirical analysis of bug reports and bug fixing in open source android apps. In *CSMR'13*, pages 133–143, 2013.

[29] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *OOPSLA'13*, pages 623–640, 2013.

[30] Guilherme de Cleva Farto and Andre Takeshi Endo. Evaluating the model-based testing approach in the context of mobile applications. *ENTCS*, 314:3–21, 2015.

[31] FindBugs. Findbugs™ - find bugs in java programs, 2016. `http://findbugs.sourceforge.net/`, accessed 2016-02-18.

[32] Chet Haase and Romain Guy. For butter or worse: Smoothing out performance in android uis, 2012. `https://www.youtube.com/watch?v=Q8m9sHdyXnE`, accessed 2016-04-13, uploaded 2012-06-29.

[33] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, pages 92–106, 2004.

[34] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *AST'11*, pages 77–83, 2011.

[35] IDC. Smartphone os market share 2015 q2, 2016. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, accessed 2016-02-25.

[36] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI'12*, pages 77–88, 2012.

[37] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE'13*, pages 672–681, 2013.

[38] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *ESEM'13*, pages 15–24, 2013.

[39] Milan Jovic, Andrea Adamoli, and Matthias Mauswirth. Catch me if you can: Performance bug detection in the wild. In *OOPSLA'11*, pages 155–170, 2011.

[40] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *Software, IEEE*, pages 70–77, 2015.

[41] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *ICST'15*, pages 1–10, 2015.

[42] Mario Linares-Vásquez. Enabling testing of android apps. In *ACM'15*, pages 763–765, 2015.

[43] Mario Linares-Vásquez, Bogdan Dit, and Denys Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *MSR'13*, pages 93–96, 2013.

[44] Mario Linares-Vásquez, Christopher Vendome, Qi Luo, and Denys Poshywanyk. How developers detect and fix performance bottlenecks in android apps. In *ICSME'15*, pages 352–361, 2015.

[45] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE'14*, pages 1013–1024, 2014.

[46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. In *ASPLOS'08*, pages 329–339, 2008.

[47] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting and fixing performance bugs. In *MSR'13*, pages 237–246, 2013.

[48] Adrian Nistor and Lenin Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *ISSTA'14*, pages 282–292, 2014.

[49] Thanaporn Ongkosit and Shingo Takada. Responsiveness analysis tool for android application. In *DeMobile'14*, pages 1–4, 2014.

[50] Christoffer Rosen and Emad Shibab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, pages 1–32, 2015.

[51] Doug Sillars. *High Performance Android Apps*. O'Reilly Media, 2015.

[52] Spotify. About spotify, 2016. `https://press.spotify.com/se/about/`, accessed 2016-06-30.

[53] Mirosław Stanek. Androiddevmetrics, 2016. `https://github.com/frogermcs/AndroidDevMetrics`, accessed 2016-05-09.

[54] Shengqian Yang, Dacong Yan, and Atanas Rountev. Testing for poor responsiveness in android applications. In *MOBS'13*, pages 1–6, 2013.

[55] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *MSR'12*, pages 199–208, 2012.

A Tool Evaluation Survey

During my summer internship in 2015, I made the debug tool drawer that swipes in from the right in the internal Android app, along with most of the tools in it (save the glue-theme and view load time). I'm now back to do my master thesis about testing for responsiveness and view performance issues in Android apps, and was hoping to start off by getting some feedback on the existing tools and how they've been working during my time away.

1. What's your occupation at Spotify? (Title/occupation and team/squad)

   Open answer

2. Were you aware of the existence of the debug tool drawer?

   Yes/No

3. Have you used any of the tools in the drawer? (If no, feel free to skip the next few questions.)

   Yes/No

4. If yes, which tools have been most helpful/useful?

   Dropdown with alternatives for the tools

5. Motivation

   Open answer

6. What do you use the tools for?

   QA/testing purposes/Design purposes/Development purposes

7. How often would you say you use the tools or the drawer?

   Every day/Every week/Every month/Every few months/Never really...

8. Is there any tool or anything else that you think needs improving?

   Open answer

9. Are there any tools or anything else that you're missing and would like to have?

   Open answer

10. Want to help further? (My objective/goal with this thesis is to create a theoretical framework about verifying highly responsive Android apps, how to test and expose causes for jank and lag and all that bad stuff, and then using that to make even more awesome tools for you guys. I'm presenting the thesis in

early June, so there's plenty of time, for now. If you have any suggestions or ideas, or would like to help along the way, please leave your username below and I'll contact you. Or feel free to ping/email/schedule me for a fika, @elinnilsson!)

Open answer

11. Comments or thoughts or anything else you want to add?

    Open answer