



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

*Distributed  
Computing*



# Android Benchmark

Master Thesis

Gino Brunner

`brunnegi@student.ethz.ch`

Distributed Computing Group  
Computer Engineering and Networks Laboratory  
ETH Zürich

## **Supervisors:**

Pascal Bissig, Philipp Brandes  
Prof. Dr. Roger Wattenhofer

April 7, 2016

# Abstract

Every time a new phone hits the market, and often even before that, it is run through countless numbers of synthetic benchmarks that showcase its computing prowess. People rarely talk about real-world performance, and if they do, it is either anecdotal (“The phone feels very snappy”, “Only occasional hick-ups”, “Apps launch immediately”), or in the form of manual side-by-side app-launching comparisons on YouTube. At the time of writing, there are, to the best of our knowledge, no real-world performance benchmark applications on Google Play.

We present the first such application, called *DiscoMark*, which is now freely available on Google Play and does not require root access. DiscoMark uses the standard *AccessibilityService* Android-API to measure the launch-times of applications; an actual real-world performance metric. Users can select which applications they want to include in the benchmark and how often the test should be run for averaging. DiscoMark then presents the average result for each app, and allows exporting more detailed results in CSV-format for further analysis. Furthermore, it allows users to compare their own results to those of our other users and gives tips on how to improve performance. DiscoMark can also be used to gain a better understanding of real-world performance and to figure out what factors influence it the most. For example, one can easily determine the effect that clearing all recent apps, or rebooting the phone has on performance.

We show results of several such experiments that were performed using DiscoMark. We found, for example, that the battery saver mode on the Samsung Galaxy S6 does not save any battery during active use, but reduces performance by about 50%. We also analysed the influence that the installing of new apps has on phone-performance, and compared the performance of different versions of Android on a Nexus 5. After a successful release and promotion, many people downloaded DiscoMark and we managed to gather data from over 5000 users and more than 1000 different phones. We gained many insights into real-world performance from analysing the collected user-data. Among other things, we discovered that uninstalling the Facebook-App brings an average speed-up of 19% and that the OnePlus One bests all other phones in real-world performance, especially when apps are already in memory (hot-starts). Overall, we found that synthetic benchmarks are a useful indicator of real-world performance, i.e., *on average*, newer and faster phones tend to perform better in the real world. However, these benchmarks only reveal a small part of the entire picture, and the benefits of real-world benchmarking are manifold.

# Contents

<b>Abstract</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Contributions . . . . .	2
<b>2 Related Work</b>	<b>3</b>
2.1 Record and Replay of User Interactions . . . . .	3
2.2 Android Benchmarking . . . . .	4
<b>3 Benchmarking Methodology</b>	<b>6</b>
3.1 Conventional Benchmarking . . . . .	6
3.2 Real-World Benchmarking . . . . .	7
3.3 Record and Replay . . . . .	7
3.3.1 RERAN . . . . .	8
3.3.2 Mosaic . . . . .	9
3.3.3 VALERA . . . . .	11
3.3.4 Android AccessibilityService . . . . .	13
3.3.5 Feasibility . . . . .	18
3.4 Application Picker and AccessibilityService . . . . .	19
3.4.1 Real-World Performance Metric . . . . .	19
3.4.2 Application Picker . . . . .	21
<b>4 DiscoMark: Real World Benchmarking Application</b>	<b>22</b>
4.1 DiscoMark Application . . . . .	22
4.1.1 UI and Functionality . . . . .	22
4.1.2 AccessibilityService Replayer . . . . .	27
4.2 Server Side . . . . .	28

<b>5</b>	<b>Lab Experiments</b>	<b>30</b>
5.1	Setup . . . . .	30
5.2	Methodology . . . . .	30
5.3	Individual App Performances . . . . .	31
5.3.1	Top 30 Applications Experiment . . . . .	31
5.4	Per-Phone Performance Evolution . . . . .	37
5.4.1	Samsung Galaxy Note II . . . . .	37
5.4.2	LG G2 . . . . .	38
5.4.3	LG Nexus 5 (5.1.1) . . . . .	39
5.4.4	LG Nexus 5 (6.0) . . . . .	40
5.5	Phone Performance Evolution - Comparison . . . . .	41
5.6	Conventional Benchmarks vs. DiscoMark . . . . .	42
5.7	OS Version vs. Performance . . . . .	45
5.8	Galaxy S6 Battery Saver Test . . . . .	46
5.9	Discussion of Results . . . . .	48
<b>6</b>	<b>User-Data Analysis</b>	<b>51</b>
6.1	DiscoMark Promotion and Statistics . . . . .	51
6.2	Rebooting Improves Performance . . . . .	52
6.3	Uninstalling Facebook and Messenger . . . . .	56
6.4	Cold and Hot Starts . . . . .	59
6.5	# Installed Apps vs. Performance . . . . .	68
6.6	DiscoMark vs. Geekbench . . . . .	71
<b>7</b>	<b>Conclusion and Outlook</b>	<b>76</b>
7.1	Conclusion . . . . .	76
7.2	OS Performance Tests . . . . .	76
7.3	DiscoMark and User-Data . . . . .	77
7.4	DiscoMark as a Service . . . . .	78
	<b>Bibliography</b>	<b>79</b>

# Introduction

---

## 1.1 Motivation

Conventional benchmarking applications to measure the performance of a given system have been around for many years. For Windows, Futuremark's benchmark suites, including 3DMark, PCMark, etc., are among the most popular and sophisticated ones. 3DMark, as the name suggests, measures the system's graphics performance. It includes several testing scenarios that focus on graphics performance, CPU physics performance and GPU/CPU stress tests. PCMark focuses on performance benchmarks related to business and home applications, such as Microsoft Office and Adobe products, as well as on storage performance and battery life. Many other benchmark applications exist that are similar to 3DMark and PCMark.

Since the rise of the smartphone somewhere around 2007, the need for mobile benchmarking applications has become apparent, and many applications now serve this need. Among the most popular currently available on Google Play are AnTuTu, 3DMark, Geekbench, Quadrant, Vellamo, GFXBench and PCMark. These apps are similar to each other, as they measure a phone's performance based on certain closed testing scenarios, including different kinds of computations in order to test different parts of the hardware. For example, AnTuTu measures performance in the categories 3D, UX, CPU and RAM. Quadrant is similar to AnTuTu, however, instead of testing UX performance it includes an I/O benchmark. 3DMark, like its PC counterpart, and GFXBench focus on graphics performance. Geekbench measures CPU performance, while Vellamo also tests browser performance. PCMark simulates real-world applications, such as Pinterest, photo-editing software and video playback, in order to provide a performance measure that is closer to actual real-world performance as felt by the user, as opposed to the artificial performance measures that most other benchmark software provides. However, even PCMark does not actually perform real-world tests, as it still only simulates real-world apps. Generally, several benchmarking apps claim to test real-world performance, while in reality, they measure *simulated* real-world performance.

Measuring actual real-world performance is difficult, and even more so due to the highly fragmented nature of the Android ecosystem: Many different versions and flavours of Android are running on thousands of phones from hundreds of manufacturers. Furthermore, every person uses his phone differently and installs different apps, and therefore will have a different user-experience. Conventional benchmarks do not factor in these individual factors, nor do they measure any real-world metrics. Thus, we cannot be sure that a high benchmark score corresponds to high performance in all situations. For example, a certain phone might have high-end hardware, but due to bad implementation of the operating system, still perform worse than similar phones. Another phone might perform as expected until a certain number of apps are installed, or a large number of background processes is running, and then suddenly experience a sharp drop in performance. A third phone might have a custom caching implementation and keep only a small number of apps in memory. These and other phenomenon cannot be detected by synthetic benchmarking applications, and we as users therefore remain ignorant, and either have to trust the manufacturer or subjective reviews when making our buying decisions. Our goal is therefore to build a novel benchmarking application that allows for actual real-world benchmarking. Using the app we can then gain new insights into real-world performance of Android phones. Since, in the end, all we care about when it comes to a phone's performance is *real-world* performance, such a benchmarking-application is much better suited to guide us in our buying decision. After all, we don't care if our brand-new phone scores highest in AnTuTu, if our old phone is still running more smoothly.

## 1.2 Contributions

In summary, our main contributions are:

1. Analysed feasibility of different approaches to perform real-world benchmarking on Android devices
2. Developed DiscoMark, a real-world benchmarking application for Android. DiscoMark does not require root-access and can therefore be downloaded and used by everyone
3. Analysed a large amount of data from the users of DiscoMark and gained many insights into real-world performance of Android phones
4. Performed a series of controlled experiments using DiscoMark that allowed us to gain further insights that could not be easily obtained from user-data

# Related Work

---

## 2.1 Record and Replay of User Interactions

Record and Replay of user interactions has been a subject of research for many years, and has been studied on various platforms, including desktop computers and Android.

Most of the existing Application/GUI-level approaches serve the purpose of application testing. Google provides several tools, among which are AndroidJUnitRunner [1], UI Automator [2], Espresso [3], UI/Application Exerciser Monkey [4] and monkeyrunner [5]. AndroidJUnitRunner is a JUnit 4-compatible test runner for Android, able to work with UI Automator and Espresso test classes. UI Automator is a testing framework that provides a set of APIs to build blackbox UI tests that perform interaction on user and system apps; it is available since Android 4.3. In contrast to UI Automator, Espresso enables the writing of white box-style automated tests, i.e., it requires the code of the app being tested. UI/Application Exerciser Monkey runs on an emulator/device and generates pseudo-random streams of events (clicks, touches, gestures, etc.), and is mainly used for stress-testing applications. Monkeyrunner provides an API for controlling a device from an ADB shell. Its main purpose is to perform application tests at the functional/framework level and to run unit test suites. The approaches mentioned above, while not specifically designed to perform record-and-replay, could be used to perform record-and-replay on Android devices. Limitations of those approaches include limited cross-device capability, lack of timing accuracy, limited capability to replay complicated touch gestures, requirement of knowledge of underlying app-code or need for external control over ADB.

Over the past few years, dedicated record-and-replay frameworks have been developed. Gomez et al. [6] introduced RERAN, a record-and-replay framework for Android that at the time of publication, was able to record and replay 86 out of the top-100 free Android apps. RERAN captures the driver-level event stream of the phone, which includes GUI and sensor events. The captured events

can later be injected by a phone-based agent and are replayed with microsecond accuracy. Limitations include the lack of cross-device capability due to driver-differences between different phones. Furthermore, RERAN does not support sensor-events, since these events are not accessible due to security issues.

Zhu et al. [7] introduce Mosaic, which extends RERAN with cross-device capability. They do so by introducing a *virtual screen abstraction* technique. Touchscreen specific input events are virtualized, and can later, through reversing the process, be replayed on another phone. Mosaic requires calibration in order to be able to translate between different devices. Furthermore, it requires changes to the Android Framework and like RERAN, is not able to record and replay sensor input. At the time of publication, Mosaic was able to record and replay 45 out of 50 tested applications.

The authors of [6] introduced their follow-up work, VALERA, which improves upon many of the shortcomings of both RERAN and Mosaic, albeit at the cost of higher complexity. VALERA is able to record and replay sensor and network input, event schedules and inter-app communication via intents. Like Mosaic, it requires an instrumented Android Framework. However, also the apps themselves must be instrumented through byte-code rewriting, in order to perform API interception.

As will be discussed in Chapter 3, we evaluate many of the mentioned record-and-replay approaches for their feasibility to be used as part of a real-world benchmarking application, but eventually end up taking a different approach. Our solution does not strive to be a full-fledged record-and-replay framework, but instead focuses only on necessary features, and we can therefore use high-level Android APIs, which in turn enable us to achieve our goal without requiring phones to be rooted and without the problems of inter-device capability.

## 2.2 Android Benchmarking

There are many benchmarking applications available on Google Play. The most popular applications include AnTuTu [8], GFXBench [9], 3DMark [10], Basemark OS II [11] and Vellamo [12]. AnTuTu and Basemark OS II are general-purpose benchmarks and test devices according to a variety of performance metrics. GFXBench and 3DMark focus on graphics performance, while Vellamo mainly tests CPU speed. These benchmarks are artificial; they use internal computations and measurements to determine the performance of a device, instead of measuring performance during real-world usage. PCMark Mobile attempts to be more representative of “real-world” usage, by using mock-up versions of Pinterest, photo-editing software and by performing video playback.

Pandiyani et al. [13] performed a performance and energy consumption analysis for a collection of smartphone applications, which they call MobileBench.



MobileBench includes benchmarks for real-world usage, including web browsing, photo rendering and video playback. Unlike above mentioned benchmark applications, MobileBench is a framework that requires an external experimental setup in order to assess performance.

Kim and Kim [11] developed AndroBench, a tool to measure the storage-performance of Android phones. AndroBench measures sequential and random IO performance and the throughput of SQLite queries. Users can tune parameters of the benchmark, which arguably allows for real-world testing of storage performances.

The mentioned benchmarking tools and methods are limited by their representation of real-world conditions, or by the need for external measurements. As will be discussed in Chapter 3, we opt for a benchmarking approach that truly measures a real-world performance metric. Furthermore, we require our framework to be self-contained, i.e., a simple app that can be used without external measurements and instrumentation.

# Benchmarking Methodology

---

In this chapter we discuss the feasibility of different approaches to benchmarking by performing record-and-replay. Section 3.1 gives an overview of conventional benchmarking techniques for mobile phones, and explains why we need to go into a different direction for this thesis. In Section 3.3 we discuss the possibility of benchmarking through full-fledged record-and-replay, in the style of [7, 6, 14], where we would record the usage pattern of a user, and later replay it on another device and measure the performance difference. We will also present our own approach for record-and-replay using the Android AccessibilityService. Section 3.4 introduces our final solution to real-world benchmarking.

## 3.1 Conventional Benchmarking

Conventional benchmarking applications (AnTuTu, Geekbench, Vellamo, etc.) are designed for controlled repeatable tests, since the test cases are closed scenarios within the benchmarking application. Therefore, they are good for testing the theoretical performance of the hardware, and based on that, doing comparisons between different phones. However, real-world performance is not necessarily reflected in those measurements, as the factors that influence it are too diverse and complex. For example, having dozens of apps running in the background can noticeably slow down a phone’s overall responsiveness. However, conventional benchmarks will not tell that the *current real-world performance* of a phone is slow. As they perform closed tests, which are hardly influenced by the real state of the phone, they yield almost the same results no matter when they are performed. As an example, imagine performing a conventional benchmark on a new phone. Then, some time later, when the UI of the phone is getting sluggish because too many background processes are running, you are to perform the same test again. The conventional benchmark applications will show very similar results for both situations. We tested this and the results are discussed in Section 5.6.

An important factor in phone-performance, apart from the underlying hard-

ware, is the software that runs on top of it. Android is not only highly fragmented when it comes to hardware; there are also many different versions and flavours of the Android OS, where different manufacturers apply their own changes to the Android-core. Every change to vanilla Android can potentially result in changes in performance. Furthermore, real-world performance heavily depends on the current state of the phone: (1) number of installed apps, (2) number of apps running in the background, (3) nature of the background-apps, (4) version of the OS, (5) memory-management of the OS, (6) usage patterns, (7) frequency of reboots, etc. All these factors influence real-world performance, and mostly, we can only make guesses about the influence of these factors, while making quantitative assessments is difficult and conventional benchmarking applications cannot help us here.

In the next section we define what we mean by *real-world* benchmarking.

### 3.2 Real-World Benchmarking

As explained in the previous section, conventional benchmarking applications are not adequate to represent the *current* real-world performance of a phone. Rather, they are a measure of theoretical performance, not taking into account many of the performance-affecting factors that were discussed.

Our definition of real-world benchmarking is that the performance of a phone needs to be measured based on *actual* instead of synthetic computations, or even *simulated* real-world usage. In order to achieve results that genuinely represent real-world performance, one has to work with actual applications from Google Play, and then simulate or replay a user’s interactions with those apps. During that simulation, one measures certain performance metrics. These metrics are by definition *real-world* metrics, and therefore, by measuring them, we can perform real-world benchmarks. There are two main-challenges that need to be solved:

1. Simulate real-world user-interaction on a phone, for example through record-and-replay
2. Measure real-world performance metrics during the simulation

In the next sections, we discuss different approaches for how to overcome those challenges and to develop a framework for real-world benchmarking.

### 3.3 Record and Replay

Arguably the most sophisticated solution for a personalized, real-world benchmark would be full-fledged record-and-replay. A user’s usage pattern is recorded,

and then later replayed on the same, or on another, phone. During the replay process, one measures certain performance metrics, which then result in a personalized, real-world performance measure. For example, by recording one’s usage pattern, replaying it on the latest phones that hit the market, and then comparing those results, one could make better informed buying decisions based on the benchmark results.

There are different levels of record-and-replay. The more accurate the replay, the higher the complexity of the framework. One could imagine only replaying the opening and closing of applications, simple button presses, more complicated gestures such as swipes and pinch-to-zoom or even sensor data and non-deterministic events. In the following, we will discuss three of the record-and-replay frameworks mentioned in Chapter 2, namely RERAN [6], Mosaic [7] and VALERA [14] in order of their publication. Additionally, we introduce a new record-and-replay approach using the Android AccessibilityService interface. Finally, we talk about the feasibility of these approaches for the purposes of this thesis.

### 3.3.1 RERAN

RERAN [6] (REcord and Replay for ANdroid) directly captures the low-level event stream of the phone, including GUI and certain sensor events. Thus, it operates at the driver-level. Since Android is based on a Linux kernel, one can simply read the `/dev/input/event*` files to capture event data. Touchscreen gestures are encoded as a series of events, without direct evidence as to the type of gesture. Figure 3.1 shows such a series of events corresponding to a swipe gesture. Since the events are captured at driver-level, they are dependent on the actual driver-implementation, which varies strongly across devices (see Figure 3.2).

RERAN uses the *getevent* tool of the Android SDK to read events from `/dev/input/event*`, thus generating a live log trace of the input events on the phone. After the logging-phase, RERAN translates the event-trace into Android input events, as seen in Figure 3.1.

Initially, the authors used Android SDK’s *sendevent* tool to send single input events to the phone. In order to, for example, replay a swipe gesture, one would send all the constituting events sequentially, as seen in Figure 3.1. However, *sendevent* turns out to have a small lag, which can cause gestures to be broken up, e.g., a swipe turns into a series of presses. Therefore, RERAN incorporates its own replay-agent that lives directly on the phone. To the phone, it appears as an external agent, generating events through the phone’s input devices. RERAN also supports selective replay, i.e., one can suppress specific classes of events, such as compass or gyroscope events. For the replay to work, the phone must be *rooted*. One big caveat of RERAN is the lack of inter-device

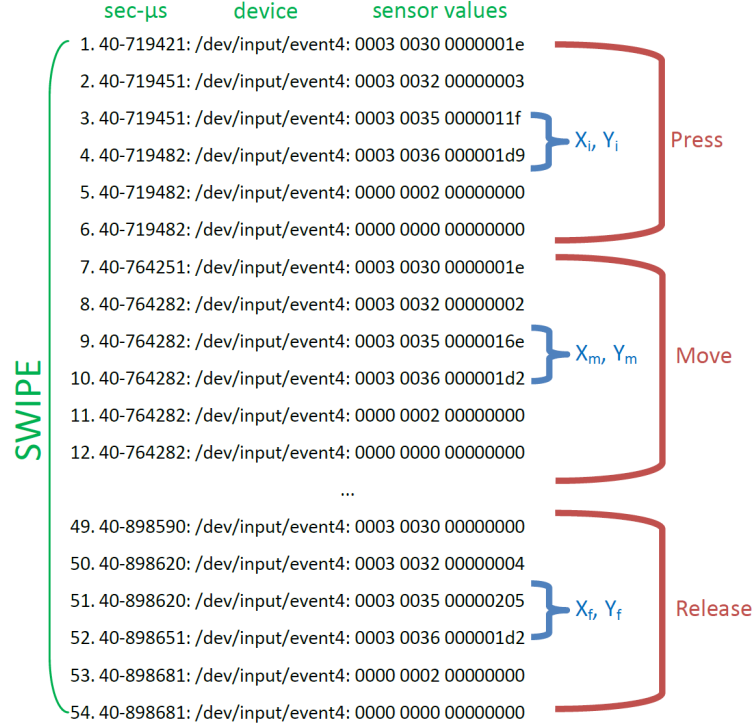


Figure 3.1: The events generated by a single swipe gesture [6]

capability. Since it captures low-level driver events, and as mentioned before, the driver-implementations vary between devices, one cannot record on device X and replay on device Y.

The authors experimentally determined the performance of their record-and-replay capabilities. They managed to replay 86 out of the top 100 apps on Google Play. Time overhead is roughly 1%, which they claim to be negligible. Trace files are also kept small and therefore have no negative impact on the phones performance during the recording.

### 3.3.2 Mosaic

Mosaic [7] is based on RERAN [6] and tackles the inter-device capability issue using a *virtual screen abstraction* approach.

The authors list several aspect that make inter-device capability difficult:

**Screen Size and Orientation** Record-and-replay of actions/events often relies on Cartesian coordinates. Also, the origin of the coordinate system

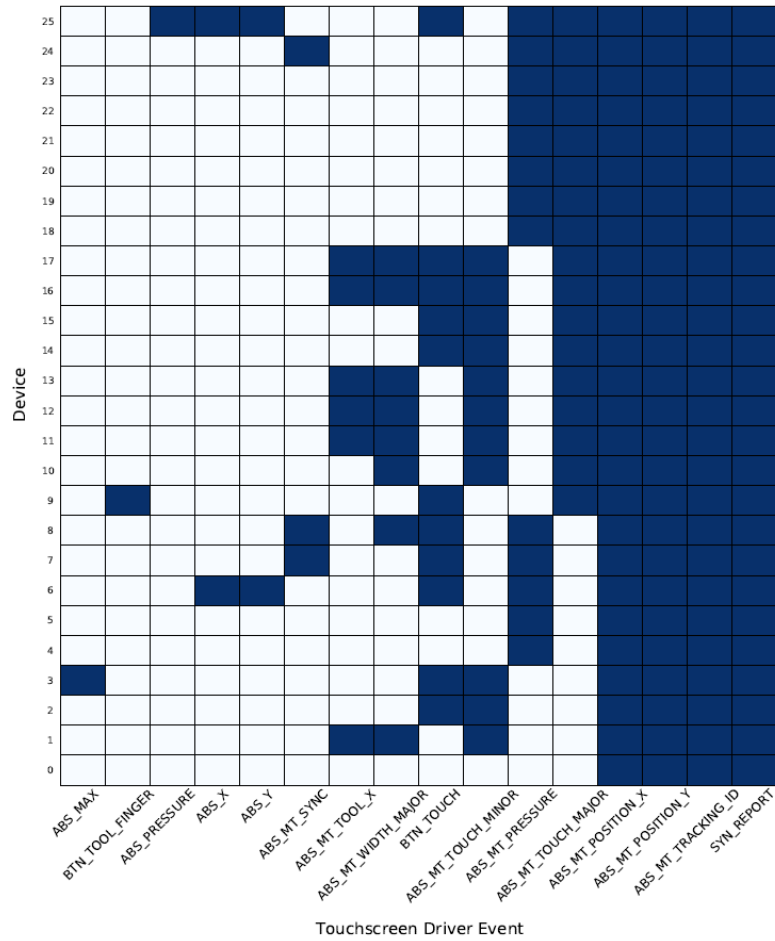


Figure 3.2: Illustration of touchscreen driver fragmentation, measured across 26 devices for *finger press* [7]

is in the top-left corner. However, after rotating a phone in clockwise direction, the origin is now in the top-right corner

**Screen Resolution** UI elements are ultimately arranged according to physical pixels, and elements can occupy different numbers of pixels on different devices

**Touchscreen Quantization** Digitalizing the user input into discrete values that can be used by the system. Touchscreen quantization values can vary between devices, and do not always correspond to their physical resolutions

**Touchscreen Driver** The touchscreen driver communicates the touchscreen's state to the operating system. Different touchscreen models communicate state in different ways. Actions like swipes are also represented differently.

**Application Framework Version** Different versions of Android can expect different events from the touchscreen driver. Also, vendors themselves can change the Android source code.

**Virtual Soft Keys** Virtual soft keys occupy screen space, i.e., they affect the touchscreen quantization

In order to overcome these difficulties, Mosaic translates real touchscreen interactions into interactions on a virtualized touchscreen. To translate from real to virtualized touchscreen interactions and vice versa, Mosaic requires a training phase, which consists of a single swipe. In addition, they use *dumpsys* to retrieve more information, such as default orientation, resolution, etc. Recording is the same as for RERAN, see Section 3.3.1, as it just uses Android’s *getevent* tool to read the event-stream. After recording, the events are virtualized from the source-phone, and then translated to fit the target-phone. Replaying is also basically the same as with RERAN, with some small additions to improve timing.

Their experimental analysis showed that they could record and replay 45 of 50 tested applications. Lag is imperceptible during recording, time overhead is less than 0.2% and memory overhead is negligible. As with RERAN, Mosaic requires the phone to be rooted, and it requires changes to the Android Framework.

### 3.3.3 VALERA

VALERA [14] (Versatile yet Lightweight Record-and-Replay for Android) is the follow-up work to RERAN, and like Mosaic, solves the inter-device compatibility problem, and also supports record-and-replay of non-deterministic events and inter-app communication. However, unlike Mosaic, VALERA ensures inter-device compatibility by performing API interception to record communication between apps and the system; this also takes care of nondeterministic events such as network events, sensor inputs and inter-app communication via intents. Furthermore, VALERA can record and replay event schedules.

VALERA consists of two parts; (1) a runtime component and (2) an API interception component. *Instrumented* apps run on top of an *instrumented* Android Framework (AF), which runs on top of an unmodified VM and kernel. App instrumentation is achieved through byte-code rewriting, and is crucial in order to intercept API calls and intents. App instrumentation is done automatically by using an off-the-shelf tool called *Redexer*. First, a *Scanner* takes the original APK file and an *Interceptor specification*, and then finds all the call-sites in the bytecode that match the specification and should therefore be intercepted. The *Interceptor generator* produces a dynamic intercepting module and a stub that is then passed to the *Redexer*, who finally performs the bytecode-rewriting and repackaging. AF instrumentation is done manually and used to log and replay the event schedule.

Interception of events and elimination of nondeterminism is achieved as follows:

**Motion and Key Events:** Main sources that drive app execution. RERAN supports complex interactions since it records event streams at the OS driver level, however, it has no knowledge of the app’s event order. VALERA records motion and key events on the app’s side by intercepting `dispatch-InputEvents`.

**Sensor Events:** (1) low-level sensors (accelerometer, gravity, gyroscope, etc.) are invoked via the `SensorManager` API, and (2) high-level sensors (GPS, Camera, audio, etc.) use upcalls and downcalls.

**Network non-determinism:** Network-traffic is intercepted and written to a log. During replay, the network-traffic is fed to the app from the log instead of from the network interface.

**Random number nondeterminism:** The random number API can also cause non-determinism. Depending on the API, VALERA intercepts just the seed, or every generated random number.

**Intercepting Intents:** VALERA intercepts `onActivityResult()` methods to obtain, e.g., an image captured by the camera

**Event Schedule:** The deterministic record-and-replay of the event schedule (simplified: which binder/looper/background thread does what and when) is slightly more involved, and we therefore refer the reader to the relevant parts of the original paper [6].

Finally, record-and-replay is achieved through auto-generated stubs:

**Stubs** API call interception is realized by redirecting the original API call to a stub. This ensures that during replay, the app uses the recorded data instead of fresh sensor data

**Timing** VALERA also records the timestamp of each intercepted method. Feeding from the log is much faster than real-world execution, therefore, VALERA has to sleep specified times in order to ensure the same speed of execution as during the original recording

**Exceptions** VALERA also records and replays any runtime exceptions.

Their experimental analysis shows negligible overhead, and performance exceeds previous approaches. Although not specifically mentioned, we assume that RERAN also requires the phones to be rooted.



Table 3.3 summarizes the important differences between the approaches presented above. We can conclude that RERAN is not feasible for our purposes, since it cannot record on one device and then replay on another. Mosaic and RERAN could in theory both be used to measure a phone’s performance during replay, since they are, to the best of our knowledge, the most full-featured record-and-replay frameworks for Android. Thus, they can basically record and replay everything (with small differences), and therefore, one could measure many different metrics to determine a phones performance. There would definitely be issues to be resolved, especially when it comes to finding suitable performance metrics, and the timing of events. For example, recording on a brand-new, high-end phone and then replaying on a 2-year old, very slow device, could lead to problems.

Unfortunately, RERAN is also the only framework where the source code is available. Furthermore, as they are more capable than RERAN, Mosaic and VALERA have more complex implementations. So, while very powerful and in theory capable of performing personalized, real-world benchmarks, re-implementing either Mosaic or VALERA ourselves does not lie within the scope of this thesis. Furthermore, the strict root-requirement poses restrictions on the possible user-base, and would make crowd-sourced data collection even more difficult. Also, in Section 3.3.4 we will show that we do not need such a powerful record-and-replay framework, but can instead rely on a standard Android API, which in itself has many advantages.

	RERAN	Mosaic	VALERA
requires root	y	y	y
inter-device capable	n	y	y
requires app instrumentation	n	n	y
requires changes to AF	y	y	y
replay non-deterministic events	n	n	y
source code available	y	n	n
overall ranking	3rd	2nd	1st
overall implementation complexity	low	middle	high

Figure 3.3: Comparison of relevant aspects of the presented record-and-replay frameworks

### 3.3.4 Android AccessibilityService

Android provides an Interface called *AccessibilityService* that helps developers improve usability for users with impaired vision, hearing, etc. An accessibility service runs in the background like a normal service, and receives callbacks from the system when *AccessibilityEvents* are fired. An *AccessibilityEvent* denotes some kind of state transition in the user interface, e.g., `focusChanged`,

`buttonClicked`, etc. On receiving such an event, the accessibility service can query the content of the current window, e.g., traverse the view-tree, in order to extract useful context-information. Once the accessibility service has extracted all necessary information, it can then perform an action, such as clicking a button, sending an intent, scrolling, swiping, etc.

The obvious advantage of using a standard Android API library, as opposed to the driver-level solutions discussed earlier, is the inherent inter-device capability and future-proofness of the resulting implementation. However, we will later see that there are still differences between devices of different manufacturers, even when using such a high-level library. The goal is still to capture a user's interactions with the phone, and then to replay them on any other phone. The types of *user-interactions* we want to record-and-replay are given here:

1. Open Application
2. Close Application (Press Home Button)
3. Open and close “Recent Applications”
4. Press Back Button

As mentioned before, an `AccessibilityService` can perform certain actions that allow it to interact with a phone. The goal is to use these *Accessibility Actions* to mimic the *user-interactions*, so that they can be replayed on another phone. The most relevant *Accessibility Actions* are the following:

#### **Global Actions**

**Home** : Corresponds to pressing phone's home button

**Back** : Corresponds to pressing phone's back button

**Recents** : Open up the “Recent Apps” drawer

**Notifications** : Pull down notifications window

For a complete list and more information, see [15].

#### **Local Actions**

**Click** : Click a view element, e.g., button

For a complete list and more information, see [16].

Before we can use *Accessibility Actions* to replay *user-interactions*, we first need to record those *user-interactions*, based on the incoming *Accessibility Events*. The following is a list of the relevant *Accessibility Events*:

- `TYPE_WINDOW_CONTENT_CHANGED`: Usually fired when one or more view-elements of the current windows change
- `TYPE_WINDOW_STATE_CHANGED`: Fired when an application is closed or opened, “Recent Apps” is opened, the screen is unlocked or a dialog (e.g. app chooser) pops up or is dismissed.
- `TYPE_VIEW_CLICKED`: Fired when a button, or any other view element, is clicked
- `TYPE_VIEW_SELECTED`: Fired when an App is started from the “Recent Apps” spinner
- `TYPE_VIEW_FOCUSED`: When a view-element, like a text field, gains focus

There are more types of events, but these five were determined to be the most relevant ones based on experiments where we analyzed the types of events that are fired during different usage scenarios. An excerpt of those experiments can be seen in the Tables 3.4, 3.5. For certain Actions, e.g., opening an application, we recorded the Accessibility Events that were fired. We also extracted attributes from the events themselves, namely *PackageName*, *ClassName*, *Text*, *ContentDescription*. *PackageName* and *ClassName* correspond to the activity that caused the event, e.g., the App being started, or the Android launcher. *Text* and *ContentDescription* are attributes that give additional information, such as the name of the view-element that was clicked. Under *misc.* we listed other interesting attributes when applicable, such as the position of the view-element that was selected.

**Open Application “My Files”**

Nr.	EventType	PackageName	ClassName	Text	ContentDescription	misc.
1	VIEW CLICKED	com. sec. app. launcher	android. widget. TextView	[My Files]	My Files	
2	WINDOW STATE CHANGED	com. sec. an- droid. app. my- files	com. sec. an- droid. app. my- files. MainActiv- ity	My Files	null	
3 (some- times)	WINDOW CONTENT CHANGED	com. sec. an- droid. app. my- files	android. widget. ListView		null	ItemCount: 12509

Figure 3.4: AccessibilityEvents that are fired when opening the application “My Files”

**Pull Down Notifications**

Nr.	EventType	PackageName	ClassName	Text	ContentDescription	misc.
1	WINDOW STATE CHANGED	com. android. systemui	android. widget. FrameLayout	[Notification shade.]	null	
2	WINDOW CONTENT CHANGED	com. android. systemui	android. widget. FrameLayout		null	

Figure 3.5: AccessibilityEvents that are fired when pulling down the notifications window (*swipe gesture*)

One important thing to realize from these user-interactions and their corresponding *Accessibility Events* is that interactions like swipes and pinch-to-zoom do not fire any events, and can therefore not be recorded (see the lack of a “swipe event” in Table 3.5). This fact alone means that full-fledged record-and-replay is not achievable with the AccessibilityService API, and we therefore need to focus on more basic interactions, such as the opening and closing of applications.

However, even for relatively simple user-interactions, the translation from *Accessibility Events* (generated by user-interactions) to *Accessibility Actions* is not trivial. It is made difficult by the large number of different kinds of interactions (open/close apps, swipe, scroll, etc.) and by differences between the AccessibilityService-events across different phones. Furthermore, many different user-interactions can produce the same event-sequences, which are then hard to distinguish reliably.

We implemented a proof-of-concept in order to better assess the feasibility of this new approach. In the following, we will briefly describe the three main building blocks of the PoC Android app: The recorder service and the replayer service.

### Recorder Service

The *recorder* starts listening to *Accessibility Events*. As long as the recorder is running, incoming *Accessibility Events* are written to a buffer that is then used to match the *Accessibility Events* to *Accessibility Actions*. The matched actions are then encoded for the replayer. The replayer needs to distinguish between global and local actions. A global action is always a one-step process, i.e., “go to home screen”. A local action on the other hand can consist of several *micro actions*. Micro actions are the steps that need to be done before, e.g., actually clicking on a button. For example, before we can click a button, we first need to obtain a reference to it by traversing the view tree. Therefore, the micro actions need to encode the necessary steps for the replayer to successfully perform these steps. Micro actions are not a part of the API, but a construct that we built to help us replay more complex user-interactions.

### Replayer Service

The replayer, like the recorder, is also an Accessibility Service. It receives a list of global and local actions from the recorder that it has to replay. One important aspect of record-and-replay is the timing with which actions are executed. Being a high-level library, the AccessibilityService does not allow for accurate timing. However, there are some timing-constraints that have to be taken into consideration. For example, one cannot just iterate through the list of actions and issue them as fast as possible, since the phone will not be able to keep up. Therefore, we have to ensure that an issued *Accessibility Action* terminates, i.e., has fired certain *Accessibility Events*, before we issue the next action in the list. Since we know the actions we are performing, and we also

know the kinds of events that will be triggered due to those actions, we know when a certain action has terminated. For example, if the next action in the list is “Close Application” (i.e. press home button), we know from our experiments that we expect a `WINDOW_STATE_CHANGED` event. Therefore, after issuing the global “Close Application” action, we wait for the next `WINDOW_STATE_CHANGED` event to arrive. Once it arrives, we issue the next action, and so on.

### 3.3.5 Feasibility

Using the PoC Android application, we were able to record a series (of arbitrary length) of user-interactions on one phone, and then replay it on the same, or certain other phones.

However, we quickly realized that our record-and-replay solution did not work well across all devices we tested it on, mainly due to differences in how the `AccessibilityService` worked. In the following, we list the biggest problems with the `AccessibilityService`-driven record-and-replay approach:

1. There are differences between devices when it comes to the kinds of `AccessibilityEvents` that are fired. For example, an LG G4 does not fire a `CLICK` event when the user opens an application that was placed inside a folder on the home screen, therefore, there is no way of determining that an application was opened by a click. Meanwhile, a Samsung Galaxy Pro tablet fires `CLICK` events as expected.
2. There are differences between devices when it comes to how they access the view elements of the current window. For example, an LG G4 accesses all launcher screens as one view tree, i.e., whenever a view element (e.g., an icon) is on *any* of the home-screens, it can be retrieved. The Samsung Galaxy Pro Tablet on the other hand only retrieves the window content of the currently visible home-screen. If the app-icon we are looking for is on another screen, it cannot be found without swiping left/right and searching again. However, swipes on the home screen cannot be recorded/replayed, and furthermore, a left/right swipe cannot be performed using the `AccessibilityService`.
3. Certain functions (e.g. `getRootInActiveWindow()`) return different values on different phones. On the LG G4 for example it always returned a widget that was placed on the home screen, instead of the home screen itself, and therefore made view tree traversal impossible.

In order to accurately replay the opening/closing of applications, the first item is a deal-breaker. However, there is a workaround for this; we can always start an `Activity` using an intent, even though the user actually clicked on it

(given we can figure out the class and package of the activity that was started, which we usually can by inspecting the `WINDOW_STATE_CHANGED` events). The problem with this approach is that it is not consistent across devices, and we are not accurately replaying what the user did. In other words, on some devices we can replay all the clicks the user made, i.e., we can start all applications by performing `CLICK` actions. On other devices, all or most applications have to be started using intents. This makes it difficult to compare results across devices.

These are the main reasons why we decided to abandon the record-and-replay approach in favour of something more reliable, as described in the next section. An important goal of ours is to make the benchmarking-application platform-agnostic, that is, it has to work regardless of phone model and manufacturer. Unfortunately, as we have seen after testing our PoC on only a handful of phones, there are too many uncertainties that we cannot control. However, we keep many of the parts that were described in this section, and basically only replace the recording-part.

### 3.4 Application Picker and AccessibilityService

This section introduces our final approach to real-world benchmarking. Many of the basics have already been covered in Section 3.3.4, and we will therefore focus on the new parts. The actual implementation of the Android application will be covered in Chapter 4. Section 3.4.1 introduces our real-world performance metric and shows how we measure it. Section 3.4.2 explains how we use an *Application Picker* in order to get rid of the problematic recorder-part, while still keeping the personalization-aspect of our benchmark.

#### 3.4.1 Real-World Performance Metric

Our benchmark is based on following real-world metric:

**Application Launch Time** : During the simulation of user-interactions we measure how long it takes for applications to open

The reasons for choosing this metric are mainly the relative ease of implementation, reliability and representativeness of real-world use. Starting an application is something we do very often on our phones, and any slow-down is immediately noticeable. Also, it is very well defined since there are only very few ways to start an application. Furthermore, we can focus on the simulation of a small, well-defined set of user-interactions and ignore intra-app interactions (pressing buttons, typing text, etc.), which helps us circumvent many of the difficulties presented in Section 3.3.5.

In order to measure the launch time of an application, the *replayer* logs information about the started application together with a timestamp. Furthermore, it also records the relevant *AccessibilityEvents* that can tell us when an application has started. The basic functionality of the *replayer* is still like described in Section 3.3.4: It issues actions from an action-list, waits for the expected *AccessibilityEvent*, and issues the next action. All while logging relevant information, that will later be used by the *benchmarker* to compute the launch-times.

There are two kinds of events that we can use to determine when an application is fully launched:

1. `STATE_CHANGED`
2. `CONTENT_CHANGED`

Before moving on, we need to decide how exactly to measure the launch-times of applications. To that end, we recorded a video of a phone opening and closing a series of apps using an early version of our benchmarking app that already included an application picker. We then computed the pixel-difference between frames using the Manhattan Norm:

$$||x||_1 := \sum_{i=1}^n |x_i| \quad (3.1)$$

Plotting the difference between frames allowed us to accurately determine when an app visibly starts and finishes loading. We can then compare these times to the timestamps of the *AccessibilityEvents* that were generated.

Figure 3.6 shows the changes between frames in form of the Manhattan norm of the pixel-differences between frames. Every spike belongs to an app being launched, and clearly depicts start and end of the app-launch.

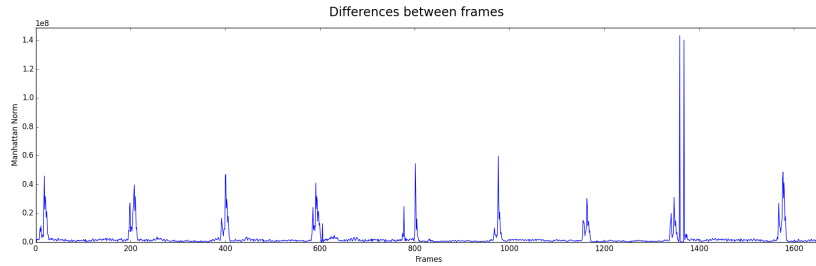


Figure 3.6: Manhattan Norm of the difference between frames plotted against the frame number.

For all the app-starts, there was a `CONTENT_CHANGED` event that was close to the end of each respective peak, and always closer than the `STATE_CHANGED`



event, which comes a little bit earlier. However, it was not always the last `CONTENT_CHANGED` event, and there was no easily discernible pattern, or a way of figuring out which of the `CONTENT_CHANGED` events to choose for measuring the launch-times. Furthermore, the `CONTENT_CHANGED` events are less reliable, because the performance of loading view-elements heavily depends on the content (number of pictures in the Gallery app), or the speed of the Internet connection (e.g. Facebook, Twitter, and many more). Since almost all apps exhibit either one or both of the just discussed dependencies, measuring launch-times by using `CONTENT_CHANGED` events incorporates many uncertain influences that we cannot account for. Another significant problem with `CONTENT_CHANGED` events is that they are not always fired, or multiple of them are fired, and they can be fired by almost anything: Alerts, incoming messages, toasts, advertisement banners, etc. Some of those cases can be dealt with by inspecting the class and package names of the activity that caused the events; if it does not correspond to the application that is currently open, they can be dismissed. However, some applications feature dynamic advertisement banners that generate events that cannot be distinguished from events fired due to the app still being launched. Other applications like Amazon Kindle and Amazon Audible generate pop-up dialogs for synchronizing the current position across devices. Those alerts fire `CONTENT_CHANGED` events and usually pop up 1-2 seconds after the applications have started, which of course causes the launch-time measurement to be wrong. Facebook for example will show that new elements arrived in your feed, which will also fire `CONTENT_CHANGED` events. These situations are hard to deal with, and are different for every app.

Due to these issues, we decided to go for the more robust solution of using `STATE_CHANGED` events as basis for our measurements. Nevertheless, we still record all the events that are triggered during the benchmark, and they are stored in the database for possible future analysis.

### 3.4.2 Application Picker

In Section 3.3.4 we have established that record-and-replay using the AccessibilityService interface is not practical. Therefore, we substitute the recorder for an *application picker*. Instead of recording usage patterns, users choose which apps they want to include in the benchmark. This also gives users more direct control and arguably offers improved functionality and user-friendliness over record-and-replay. Furthermore, as will be discussed in Chapter 5, it allows us to perform experiments in a more controlled way.

# DiscoMark: Real World Benchmarking Application

---

In this Chapter we will briefly cover the implementation of our Android benchmarking application *DiscoMark*, which is based on the groundwork that was laid in Chapter 3.

## 4.1 DiscoMark Application

DiscoMark implements the AccessibilityService-based benchmarking technique introduced in Sections 3.3.4 and 3.3.5. It provides users with the means to perform a personalized benchmark that tests the real-world performance of their phones, based on the launching of applications. In the following sections, we will cover the most important parts of the application, while omitting most implementation details.

### 4.1.1 UI and Functionality

In this section, we will explain the functions of the app while going through the different screens. The actual replay-process and benchmarking will be discussed in Section 4.1.2.

Figure 4.1 shows the home screen of the application, where a user can either choose to perform a new benchmark, look at past benchmark results or let DiscoMark analyse the last benchmark run he performed.

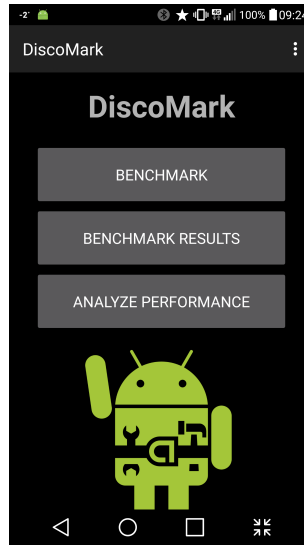


Figure 4.1: Home screen of DiscoMark.

The *Application Picker* is shown in Figure 4.2. A user can choose between all installed launchable apps. The idea is that he picks his most-used applications, representing his real-world usage pattern and thus personalizing the benchmark. Once all applications are picked, he can start the benchmark. The benchmark will be performed for a specified number of *runs*, over which the benchmark results are averaged. Therefore, choosing a higher number will result in more meaningful results, however, depending on the number of apps that were picked, the process can take a long time. During one *run*, DiscoMark cycles through the list picked applications and launches each in turn. Once it reaches the end of the list, it starts over again until the specified number of *runs* is reached. Therefore, every application will be launched *run* times.

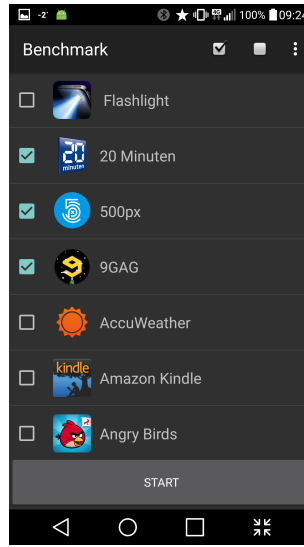


Figure 4.2: Application picker. User can pick the application he wants to include in the benchmark.

When DiscoMark finishes the last *run*, the *benchmarker* computes the results, which are then displayed to the user as can be seen in Figure 4.3. For each application that was included in the test, the average launch-time is displayed. On the bottom of the screen, the number of runs that were performed, as well as the average time per run is shown. Users have the possibility to replay this exact benchmark at any point in time, such that they do not have to re-pick the applications. This is especially useful when doing controlled tests, for example, compare the performance before and after a reboot. Users also have the possibility to export detailed results as CSV-files for further analysis. For every tested app and for every run, the app exports the launch-time. Therefore, if there were any errors that caused outliers, one can easily remove them. Furthermore, apps start slowest when they are not already in memory (cold-start) and the launch-times then gradually get shorter over the next few runs (hot-starts), which can skew the results if one just looks at averages. Exporting the results in CSV format makes it possible to separate hot- and cold starts and analyse them separately.

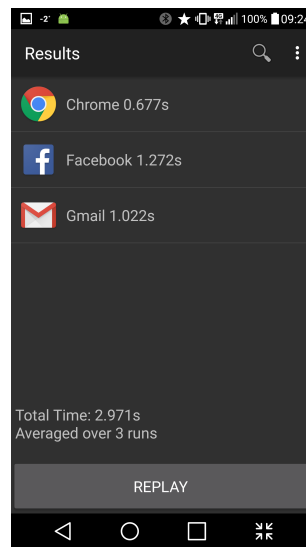


Figure 4.3: Result activity. For each tested application, DiscoMark displays the average launch-time. It also displays the number of runs, and the average time per run.

When a user clicks on one of the application result-rows as seen in Figure 4.3, DiscoMark shows a new screen, where the performance for a particular app is compared across all users in our database. This is shown in Figure 4.4.

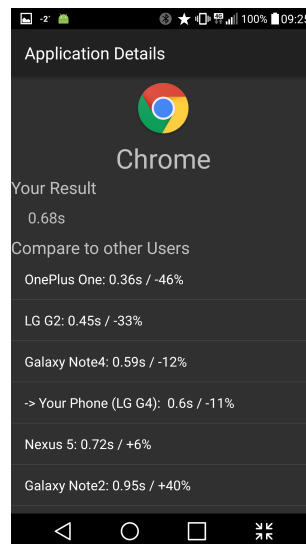


Figure 4.4: For each tested application, DiscoMark retrieves numbers from other users from the server and compares them to your own result.

DiscoMark can analyse, and compare to other users, the results of the *last* performed benchmark. We only use the last performed benchmark because it best represents the current state of a phone. A possible result of the analysis can be seen in Figure 4.5. First, we indicate whether the phone performs normally when compared to the same phone from other users. Then we compare its performance to that of other phones. Finally, if available, we provide some tips on how to improve the performance.

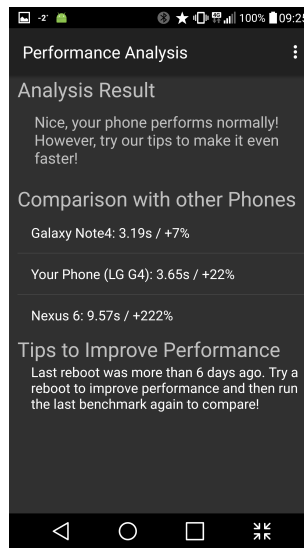


Figure 4.5: Based on the last benchmark performed, DiscoMark performs an analysis of the phone's performance by comparing the result to that of other users.

Finally, Figure 4.6 shows the settings-screen of the application, which lets users change the number of runs the benchmark will perform.

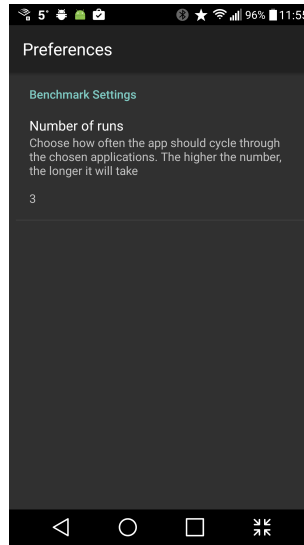


Figure 4.6: Settings screen. Users can choose for how many runs they would like to perform the benchmark.

#### 4.1.2 AccessibilityService Replayer

Most of the functionality of the replayer was already discussed in Section 3.3.4.

The application picker packs the picked applications into a special format, and encodes the necessary actions. The replayer then cycles through those *Accessibility Actions* for the specified number of runs. In other words, it starts one application after the next until the specified number of runs is reached. While doing so, it writes important information, such as *Accessibility Events*, package/class names, timestamps, etc. to a log file. The log file is used by the *benchmarker* to compute the results, display them to the user and upload them to the server. As explained in Section 3.3.4, the replayer itself is an *AccessibilityService*, and can therefore react to *AccessibilityEvents*. So, when the replayer starts a new application, it knows that there will soon be a *STATE\_CHANGED* event fired by the application that was just started. As soon as it sees that event, it can start the next application in the list. However, we are not only interested in the timestamps of the *STATE\_CHANGED* events, as explained earlier, but also in the view-elements that are loaded after. The loading of view-elements then often fires *CONTENT\_CHANGED* events, which we also want to write to the log file. Therefore, we need to make sure that we give the applications enough time to finish starting before we go on. Through inspecting our experimental results, we determined that waiting for approximately five seconds after the *STATE\_CHANGED* event arrives works well.

After the replayer finishes, the *benchmarker* analyses the log files gener-

ated by the replayer and computes the benchmark results. The benchmarker then generates a couple of result-files that are subsequently used to display the benchmark-results to the user, and to upload all the data to our server. For the upload to the server, the data is converted into JSON and then asynchronously uploaded. In order to distinguish data from different users, a hash is generated for every phone and sent to the server, together with the benchmark-results and more information about the user's phone.

## 4.2 Server Side

The backend of our application is implemented with Django,<sup>1</sup> and we use MySQL as our database engine. The server reacts to POST requests from the Android application, and takes/returns data formatted as JSON. The backend consists of the following models and views:

### Models

**BenchmarkResult** : Contains data concerning each benchmark-run: Unique user ID (hash), uptime, number of runs, os version, app version, phone model, phone manufacturer, processor model, RAM, resolution, memory threshold, number of installed apps, number of tested apps

**SingleAppResult** : For each app in a benchmark-run we create an instance of SingleAppResult that contains the application name, package/class names and launch-time information for each run. It is linked to BenchmarkResult through a foreign-key relationship.

**LogFile** : We upload the raw log-files. Therefore, if we detect an error/bug along the way, we can go back and re-compute the results to perform our analysis. The log files are linked to BenchmarkResult through a foreign-key relationship.

**InstalledApps**: A list of all installed apps. This information might be useful later when we analyse the performance across our user-database. This will allow us to slice the dataset based on one or more apps that a user has (or has not) installed. This model is linked to BenchmarkResult through a foreign-key relationship.

**BackgroundApps**: A list of applications that were running in the background before and after the benchmark. This model is linked to BenchmarkResult through a foreign-key relationship.

---

<sup>1</sup><https://www.djangoproject.com/>



In the next chapter we present the results of our lab experiments using the actual application that was introduced in this chapter, and present the insights gained from them.

# Lab Experiments

---

In order to get a better understanding of how phones perform and what the factors are that influence performance the most, we used DiscoMark (as outlined in Chapter 4) and performed a series of experiments. With the insights gained from those experiments, we were able to make well-informed decisions as to how we shall proceed, and as to what improvements we will make to the Android application.

## 5.1 Setup

Figure 5.1 depicts our experimental setup. We included three different phones, the Samsung Galaxy Note II, the LG G2 and the LG Nexus 5. In the case of the Nexus 5, we performed the experiments twice, once with Android 5.1.1, and once after upgrading to Android 6. The LG G2 and Nexus 5 have very similar specifications, while the Note II is roughly one year older. In fact, according to AnTuTu and PCMark, the Note II is expected to perform roughly 30-50% slower than its rivals.

## 5.2 Methodology

For most of the experiments, we performed our benchmark with 29 popular apps and chose the number of benchmark-runs to be 15. We then gradually installed more and more apps from Google Play’s top-apps list. Each time we installed new apps, we ran the same benchmark again. We started with only those 29 (we will call it “Top 30” from now on) apps installed (in addition to pre-installed apps), and then in each iteration we installed 20 new apps, until we reached approximately 160 additional apps.

Before each run of the experiment, we initialized every newly-installed app by running the benchmark with *all* applications picked, i.e., we launched every application on the phone once. This was to make sure that the newly-installed




		
<b>Samsung Note II</b>	<b>LG Nexus 5</b>	<b>LG G2</b>
OS: 4.4.2	OS: 5.1.1 & 6.0 Vanilla	OS: 5.02
CPU: Samsung Exynos 4412, 4-Core, 1.6GHz	CPU: Snapdragon 800, 4-Core, 2.26GHz	CPU: Snapdragon 800, 4-Core, 2.26GHz
RAM: 2GB	RAM: 2GB	RAM: 2GB
Antutu: 24,950	Antutu: 33,150 (+32%)	Antutu: 38,749 (+55%)
PCMark: 2822	PCMark: 3819 (+35%)	PCMark: 3738 (+32%)

Figure 5.1: Setup of our experiment. Three different phones and four versions of Android were used.

applications had the chance to initialize, and potentially start their background-processes. Furthermore, we tried to keep the parameters of the experiment the same for all phones to ensure “fairness” between the phones. For example, we did not deliberately restart any phones during the experiment runs. However, as will be discussed shortly, some of the phones basically stopped working and had to be rebooted, which led to interesting observations. Prior to the first experiments, we reset all phones to their factory settings. We placed all phones very close, and at the same distance, to a WiFi access point, in an attempt to eliminate performance-differences due to unfair network factors.

## 5.3 Individual App Performances

### 5.3.1 Top 30 Applications Experiment

In the first iteration of our experiment, we performed the benchmark for the top 30 apps. Figures 5.2, 5.3 and 5.4 show the results for this experiment, broken down to a per-app level. So, for each app that was included in the

benchmark, we can see how long it took to launch over the course of 15 runs of the experiment. We then proceeded to install more and more apps, for a total of 10 different experiments. We will not show all subsequent plots, but rather show the aggregated results in Sections 5.4 and 5.5. In order to give the reader an understanding of the kinds of plots we created for each experiment (Top30+0, Top30+20, etc.), we show the plots for the first experiment *Top30+0*.

On the x-axis we have the apps that were included in the benchmark, and for each app, we see the launch-times over 15 runs in form of boxes that show the standard deviation and variance. We see that many apps show similar relative launch-times on all three phones, and most have small variance. A few apps, such as Dropbox, Facebook, Messenger, Snapchat and Amazon Kindle exhibited stronger fluctuations in our tests.

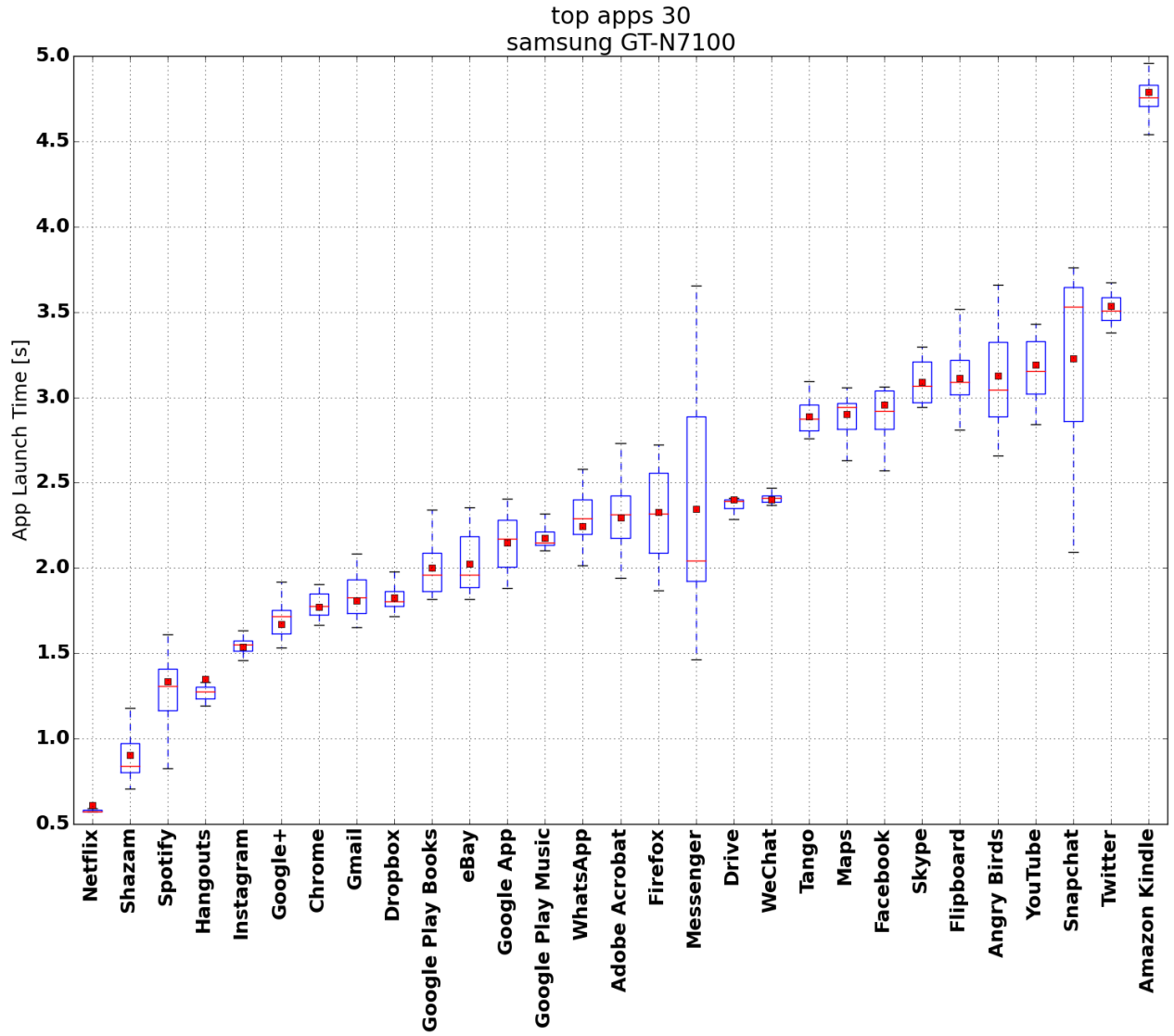


Figure 5.2: Results of top 30 experiment for Galaxy Note II

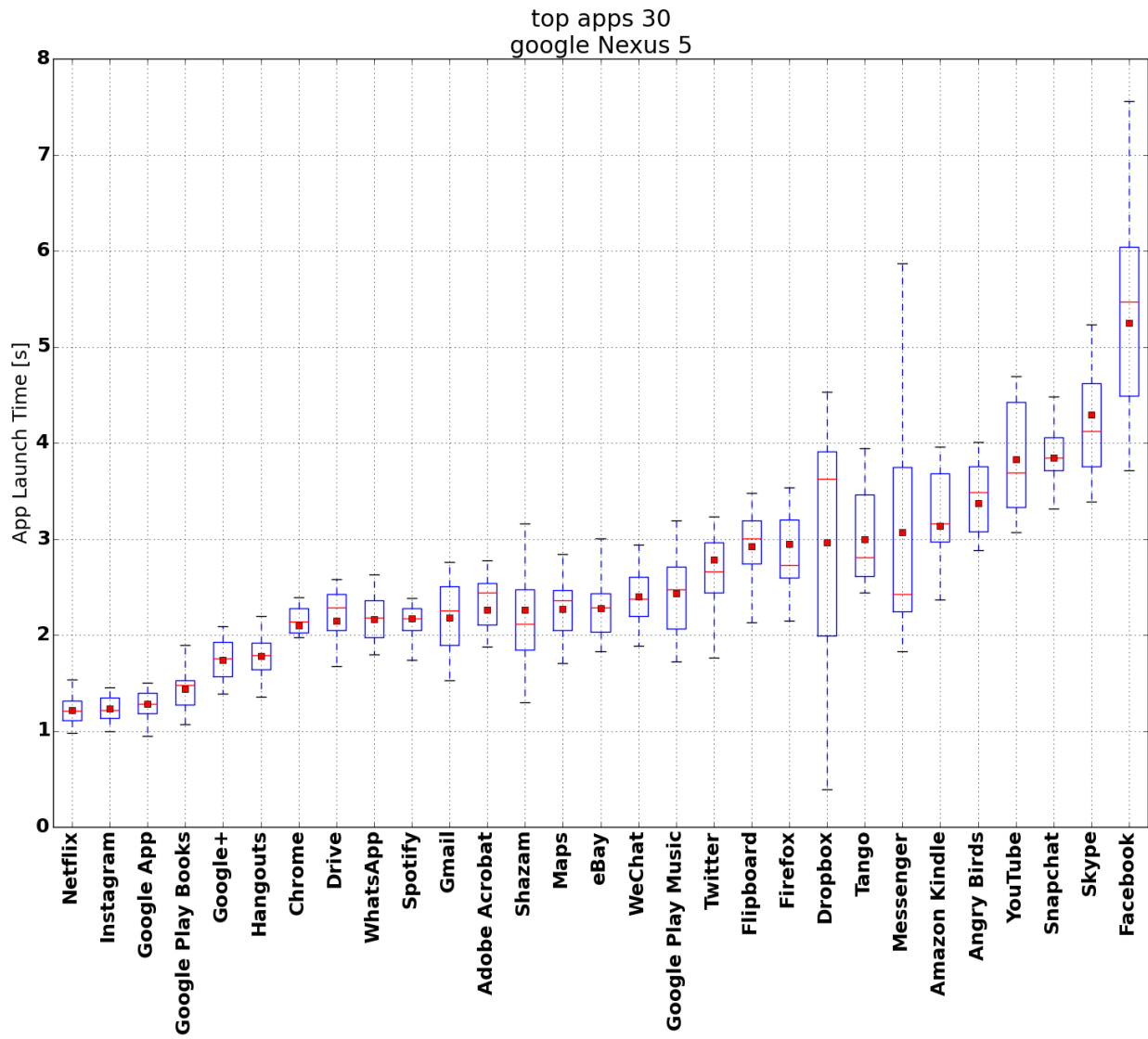


Figure 5.3: Results of top 30 experiment for LG Nexus 5

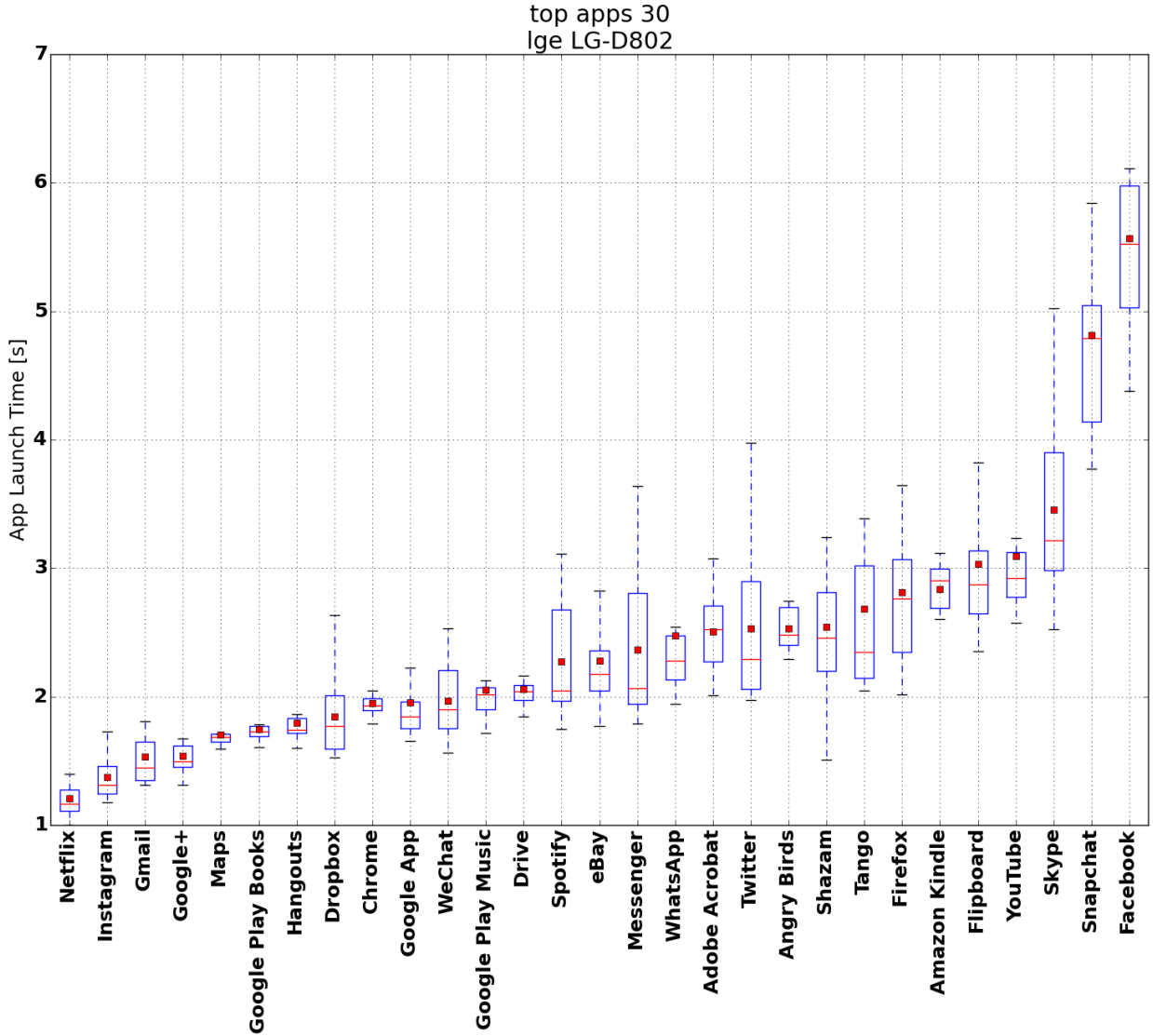


Figure 5.4: Results of top 30 experiment for LG G2

In order to make it easier to compare the three phones on a per-app basis, we created bar diagrams for each experiment. Figure 5.5 shows a comparison between the *average* launch time of each individual app for the tested phones. The x-axis is sorted in ascending order by the red bar (Galaxy Note II). We can see that there is an upward trend to the right, i.e., all three phones' launch-times increase in similar fashion. However, there are a few outliers. Furthermore, while the Note II has faster times for Netflix, Shazam and Spotify, it lags behind for Google Play Books, Maps and Amazon Kindle. This shows that it is not enough

to just measure the launch-time of a single app in order to determine a phone's overall performance. Also, keep in mind that these plots are only from the first experiment; the remaining nine show different characteristics for some apps.

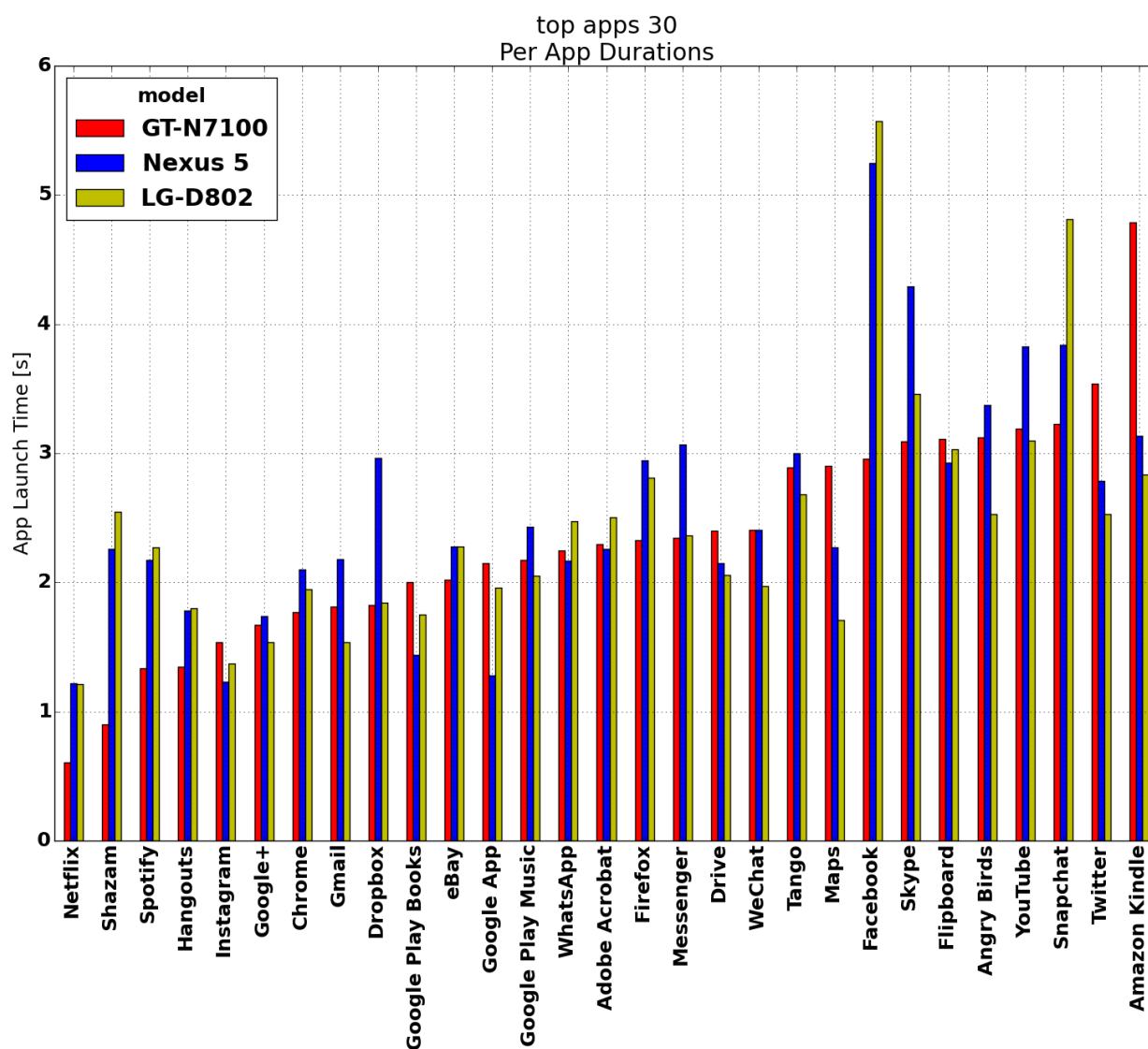


Figure 5.5: Comparison between average launch times for each app on all three tested phones (Top30+0 experiment)



## 5.4 Per-Phone Performance Evolution

As mentioned earlier, instead of showing plots of all 10 experiment-iterations, we created graphs that summarize the results for each phone.

Figures 5.6, 5.7, 5.8, 5.9 show the results aggregated over all tested apps. On the x-axis we have the experiment-iterations, where the ticks signify how many additional apps have been installed at that point. The y-axis shows the sum of the launch-times of all 29 apps. Each box consists of 15 values corresponding to the 15 runs that were performed.

Overall, there is a clear trend: More installed apps means longer app launch-times, i.e., slower performance. However, there are several oddities in the plots that warrant closer inspection.

### 5.4.1 Samsung Galaxy Note II

In Figure 5.6 we see a strictly monotonic decrease in performance until the “+140” mark, where performance suddenly increases, and variance decreases. This was because the Galaxy Note II restarted itself after the “+120” experiment. We then later restarted the Note II again, this time on purpose, and again saw an increase in performance (“+160 rest.”).

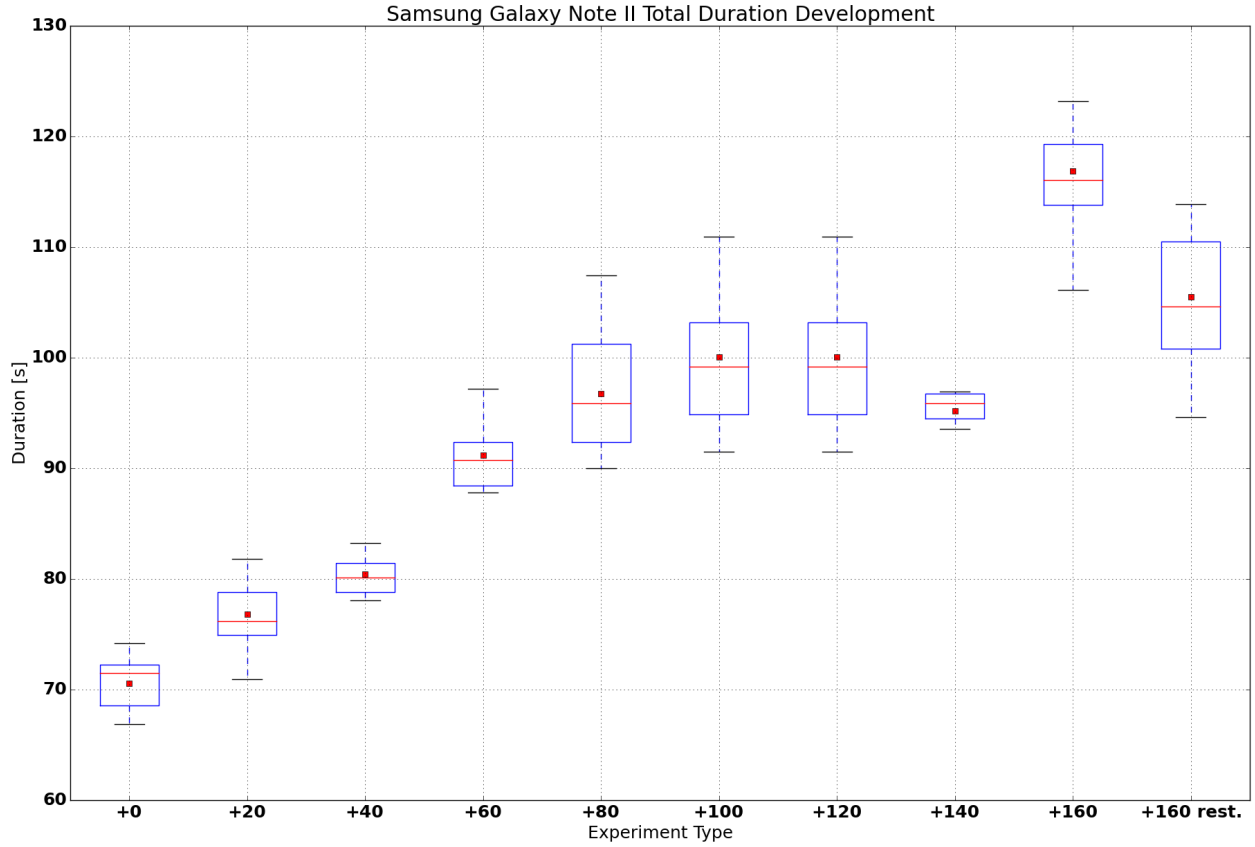


Figure 5.6: Evolution of total launch-time per run for the Samsung Galaxy Note II.

#### 5.4.2 LG G2

Figure 5.7 shows the performance-evolution of the LG G2. While the G2 never force-restarted itself, there was an anomaly at the “+100” tick. This was very likely due to updates running in the background. Usually, after installing 20 additional apps, we would first let the phones update the new apps, and only then start the next experiment-round. However, it is likely that in that case the updates were not done before the experiment, but rather were started automatically by the phone during the tests. In other instances, huge performance-decreases were observed when updates were running in the background.

Apart from that, the LG G2’s performance decreased with more installed apps, and as we can see from the last x-tick, rebooting only brought the variance

down, but did not increase average performance.

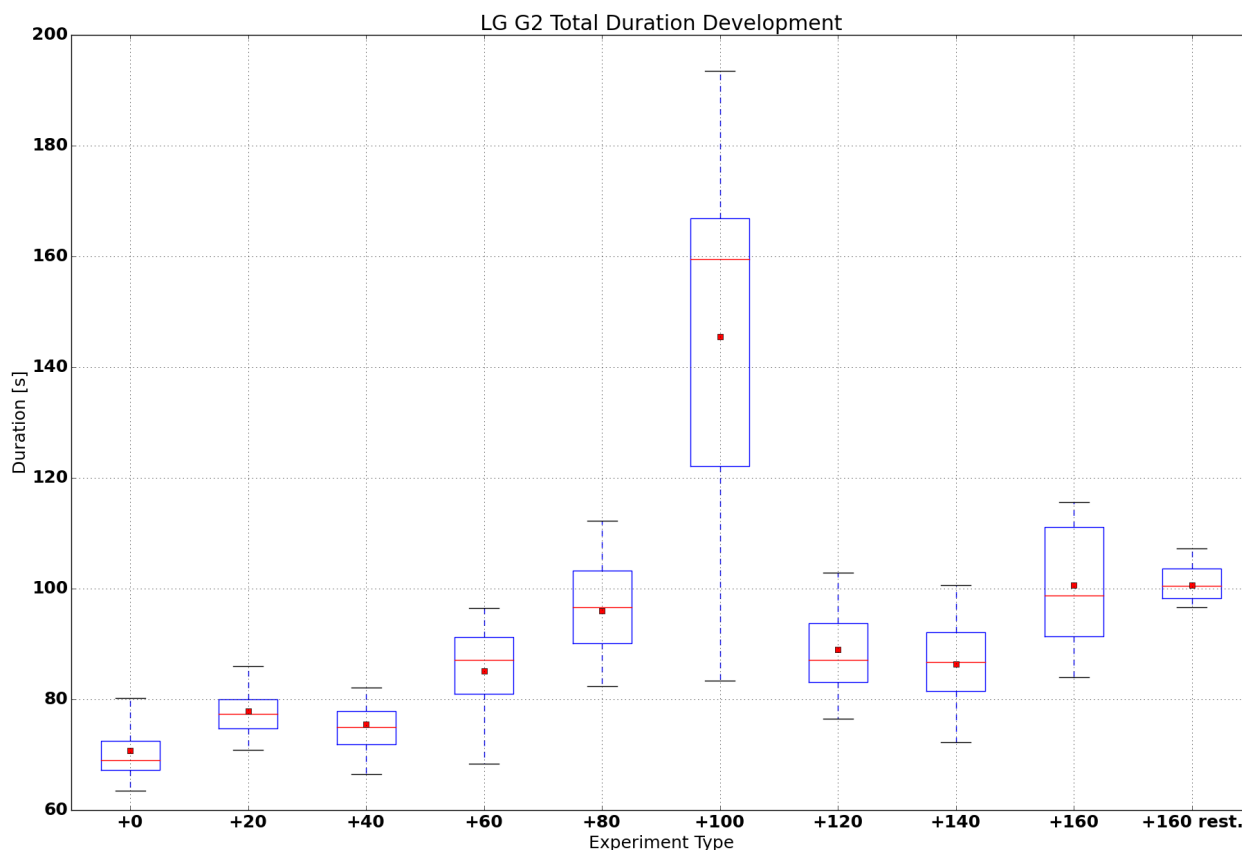


Figure 5.7: Evolution of total launch-time per run for the LG G2.

### 5.4.3 LG Nexus 5 (5.1.1)

Figure 5.8 shows the performance-evolution of the Nexus 5 with Android-version 5.1.1. This graph looks very different from what we had expected. We see several jumps that look like exponential increases, followed by sudden drops. Before each of those drops, the Nexus 5 stopped working and had to be restarted in order to continue the experiments. This graph provides more evidence that restarting your phone can greatly improve performance, especially after installing many new apps. Furthermore, it proves that there are huge differences between phones, and that these differences cannot be discovered with conventional benchmarking apps. Now we know that users with a Nexus 5 running Android 5.1.1 should be

especially mindful of frequently rebooting their phone to increase performance.

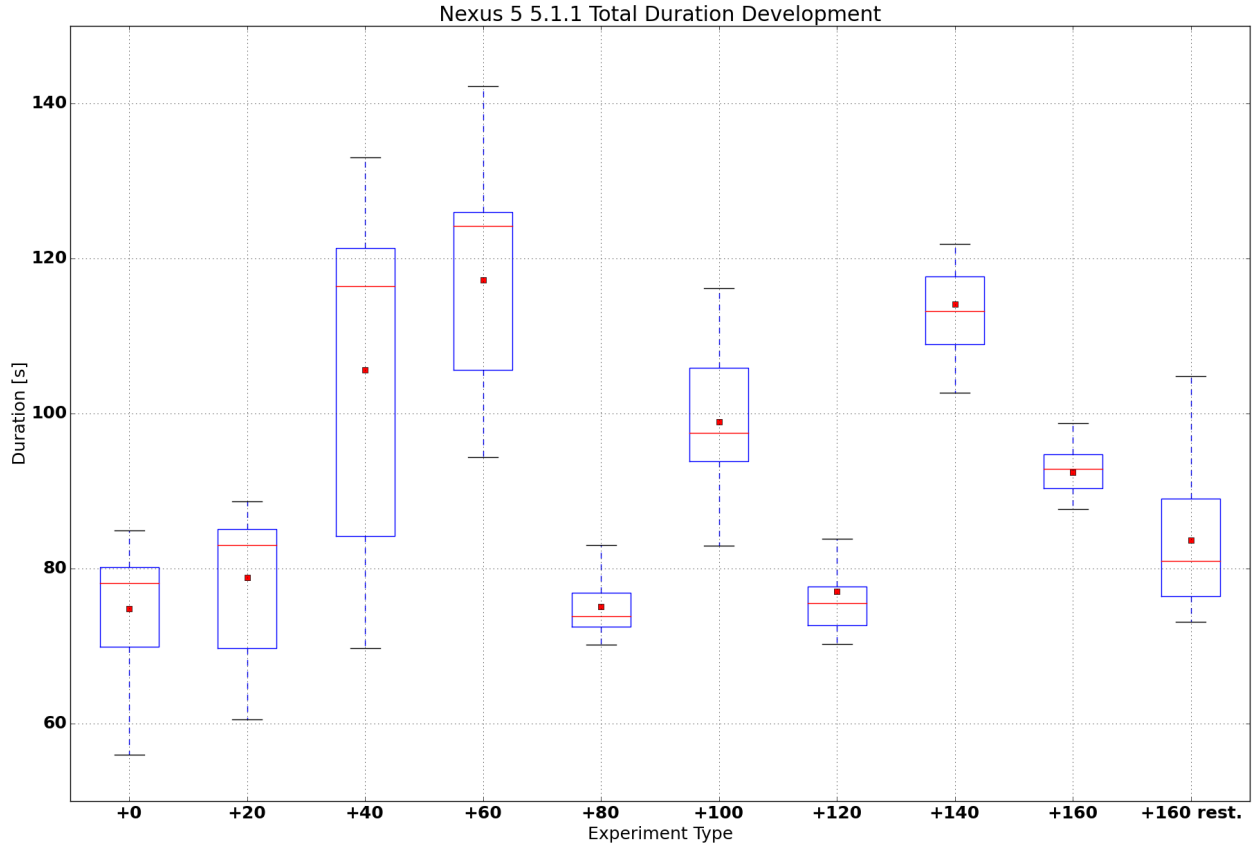


Figure 5.8: Evolution of total launch-time per run for the LG Nexus 5 with Android 5.1.1

#### 5.4.4 LG Nexus 5 (6.0)

After updating the Nexus 5 from Android 5.1.1 to 6.0, we performed the experiments again to see if the problems from before had been fixed. Figure 5.9 shows that the stability-issues are largely gone. Only once, before the “+100” test, did the Nexus 5 reboot itself, which again led to an increase in performance. Overall, the graph shows a steady decrease in performance with increasing number of installed apps.

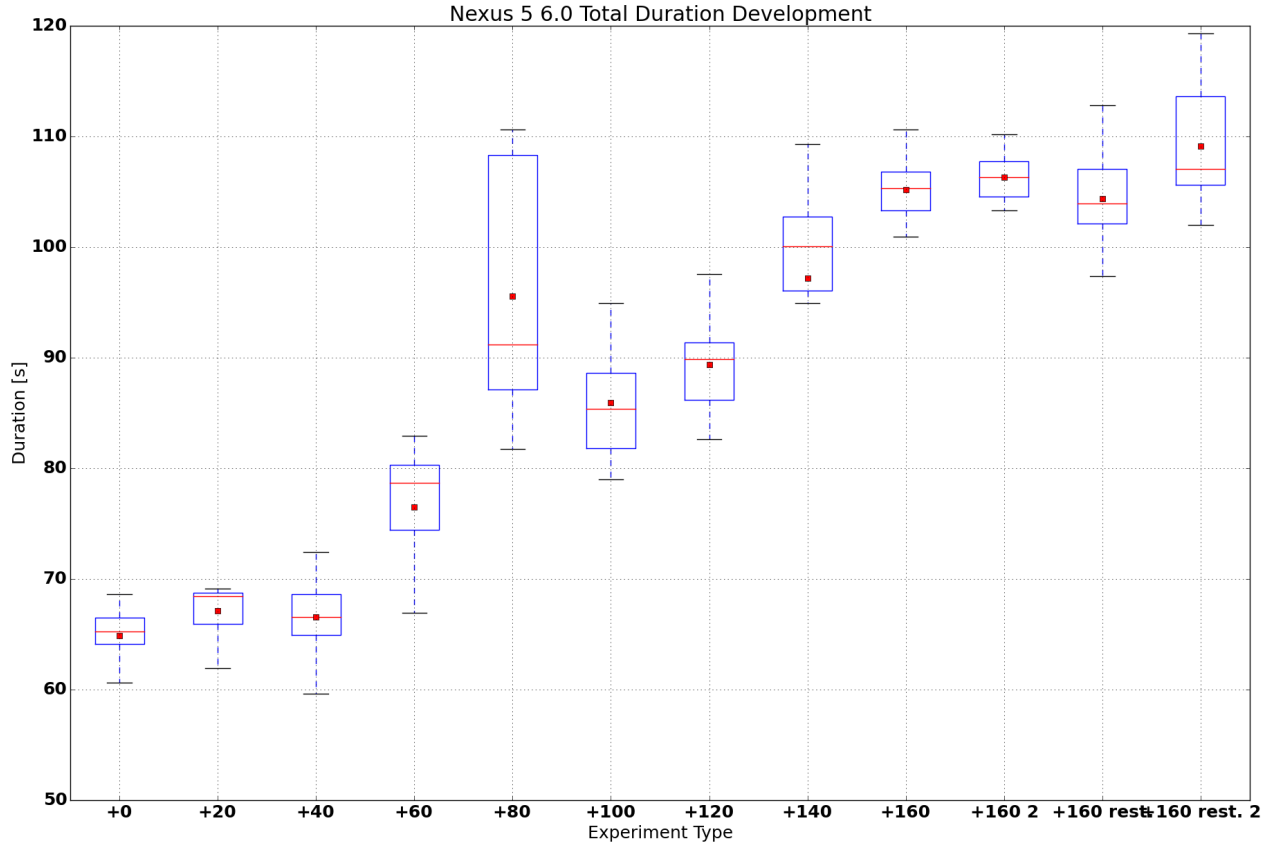


Figure 5.9: Evolution of total launch-time per run for the LG Nexus 5 with Android 6.0

## 5.5 Phone Performance Evolution - Comparison

Finally, we aggregate the plots from Section 5.4 into a single graph, shown in Figure 5.10. Overall, we see that in this experiment, the Nexus 5 (5.1.1) performed best, given frequent reboots. The Note II fared very well, considering it is the oldest phone, and it certainly performed better than the results from AnTuTu and PCMark had suggested. All phones showed a decrease in performance with an increasing number of installed apps. The black circles in the plot denote reboots, and give us an idea of how much this affects a certain phone's performance. We can certainly say that rebooting generally increases performance, and we will investigate this further based on real-world user-data. Finally, it seems that upgrading the Nexus 5 from Android 5.1.1 to 6.0 decreased *optimal per-*

*formance* (performance measured after both phones were freshly rebooted). We will investigate the performance-differences between Android-versions further in an improved and better-controlled experiment.

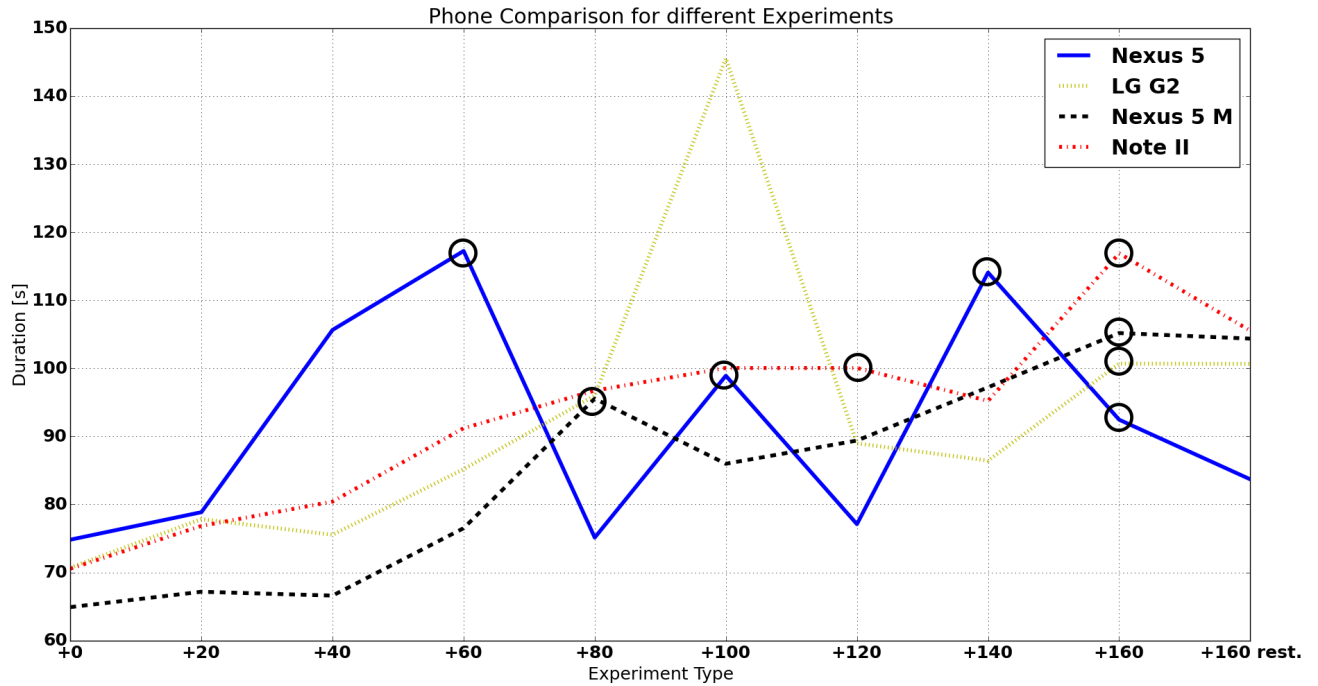


Figure 5.10: Performance evolution of all phones compared. Black circles mark when a phones was rebooted.

## 5.6 Conventional Benchmarks vs. DiscoMark

In this section, we compare the conventional benchmarking apps *AnTuTu* and *PCMark* to DiscoMark. Specifically, we want to determine how the three apps reflect the real-world performance of phones. Thus, we performed tests with all three apps in two different states for each phone: Once directly after a factory reset, and once in the “30+160”-state, i.e., the state of the last of our ten experiments where each phone had roughly 200 apps installed. We know from before that the DiscoMark-scores show a slow-down for all phones when installing more and more apps. We would expect that AnTuTu and PCMark show much less of a difference. In fact, by design, the conventional benchmarking apps should not show any difference at all, since they are made to measure the theoretical performance of a phone in an encapsulated manner, i.e., through simulation within

a closed app that should eliminate the effects of outer influences.

Figures 5.11, 5.12 and 5.13 show the differences between the three benchmarking apps. The x-axis shows the two states the phones were in, as just described. The y-axis depicts a normalized score, where the slower performance was normed to be 1. Therefore, Figure 5.11 shows that, in our experiments, DiscoMark showed roughly a 65% slowdown from the Factory reset state to the “30+160” state for the Galaxy Note II. AnTuTu and PCMark also showed slightly decreased performances, although nowhere near the value of DiscoMark. The same holds true for the LG G2 (Figure 5.12) and the Nexus 5 (5.13).

These results tell us two things:

1. The conventional benchmarking apps are not completely agnostic to the phone state, i.e., they are also influenced by how many apps are installed (more running background processes).
2. The performance-difference that AnTuTu and PCMark report is much smaller than the one reported by DiscoMark. This shows us that we cannot use existing benchmarking apps to measure real-world performance, since they do not reflect the slow-down of application-launches

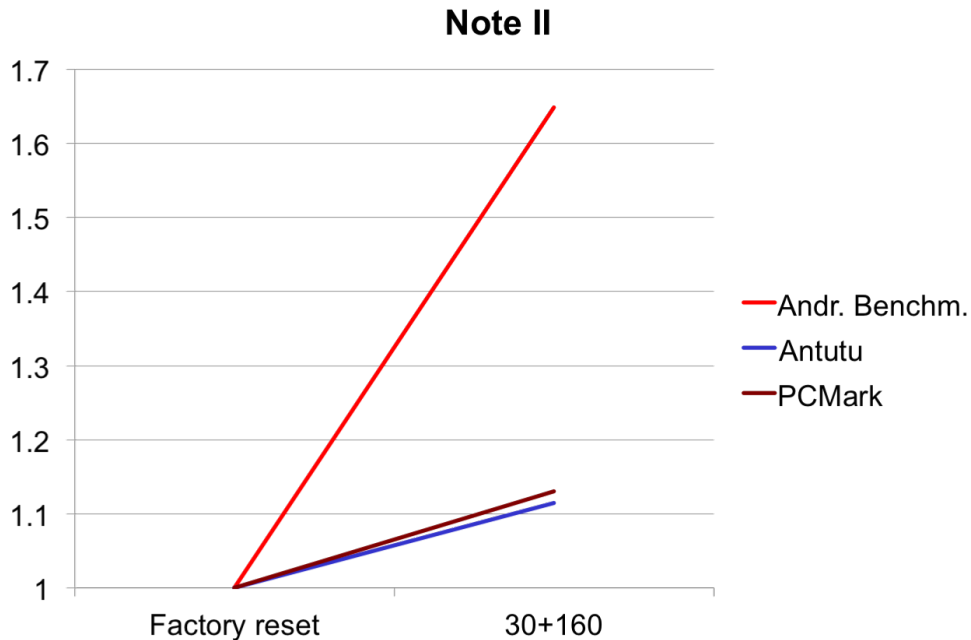


Figure 5.11: Comparison between scores of DiscoMark, PCMark and AnTuTu for the Galaxy Note II.

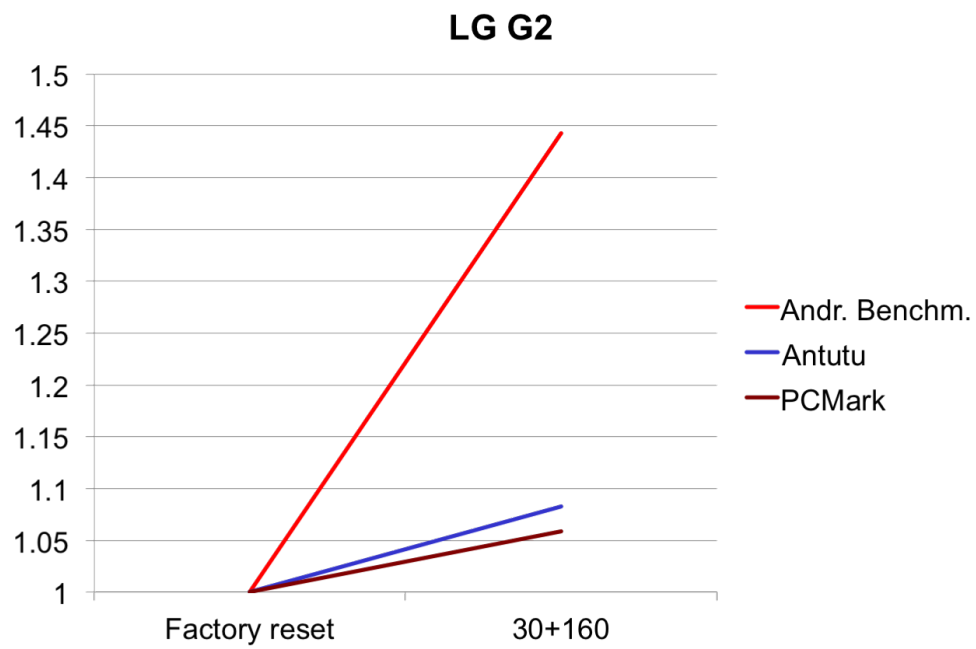


Figure 5.12: Comparison between scores of DiscoMark, PCMark and AnTuTu for the LG G2.



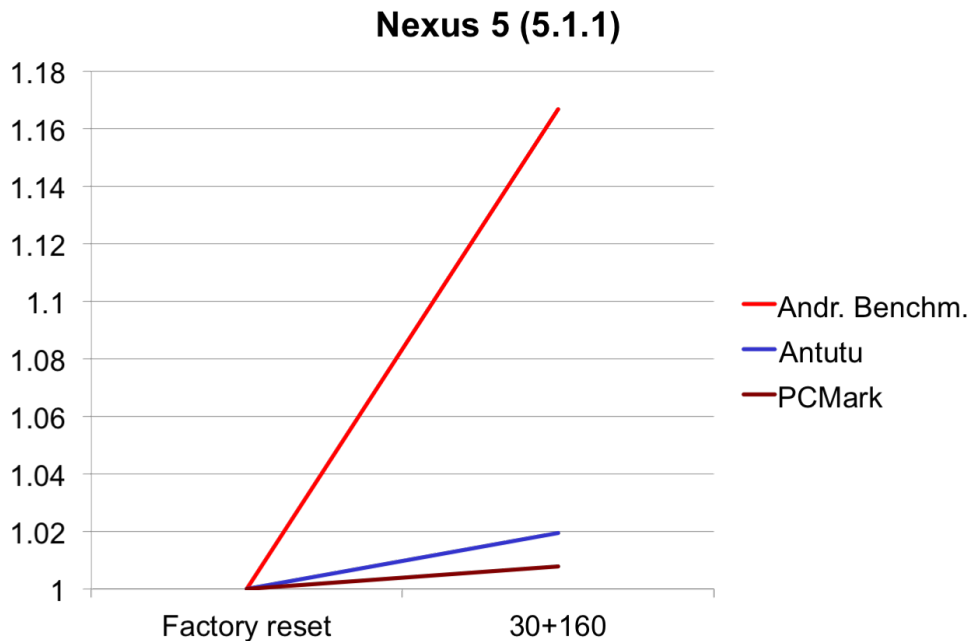


Figure 5.13: Comparison between scores of DiscoMark, PCMark and AnTuTu for the Nexus 5.

## 5.7 OS Version vs. Performance

Among Android users, Android 5.0.1 has a notoriously bad reputation due to performance problems. Many people also think that CyanogenMod will always be faster than stock Android. Furthermore, it is a common notion that phones get slower with age, i.e., with newer versions of Android. In order to verify these claims we performed an extensive series of tests on a Nexus 5 and seven different versions of Android. The results can be seen in Figure 5.14. The experiments were repeated many times to make sure the results were consistent. The y-axis shows the average application launch-time.

As expected, the performance of Android 5.0.1 is exceptionally slow compared to all other Vanilla versions. We can also see that Google did a good job of fixing the problems of 5.0.1, as the performance of 5.1.1 is very good. The same trend, at least qualitatively, can be seen between 4.4 and 4.4.4, which also was a small update functionality-wise, but focused on stability and performance instead.

Surprisingly, we see that CyanogenMod 12 (Android 5.1.1) performed very badly, especially compared to its Vanilla counterpart. This might have been a one-time thing though, since we see that CyanogenMod 11 (Android 4.4.4) almost performs on the same level as its Vanilla counterpart. However, the notion that

CyanogenMod makes your phone faster does not seem to hold, at least for the Nexus 5 with Vanilla Android. It might very well be that CyanogenMod actually performs better than flavoured versions of Android, e.g., Samsung TouchWiz. This is only speculation however, and subject of future work.

Finally, we see that performance has gradually improved (with exception of 5.0.1), and the Nexus 5 performs substantially better under Android 6.0.1 than with 4.4. Therefore, unless Google makes another 5.0.1, there is no reason not to upgrade your Nexus phone to the newest Android versions. If, after an upgrade, DiscoMark reveals that performance has decreased, one can easily downgrade again (at the cost of data loss).

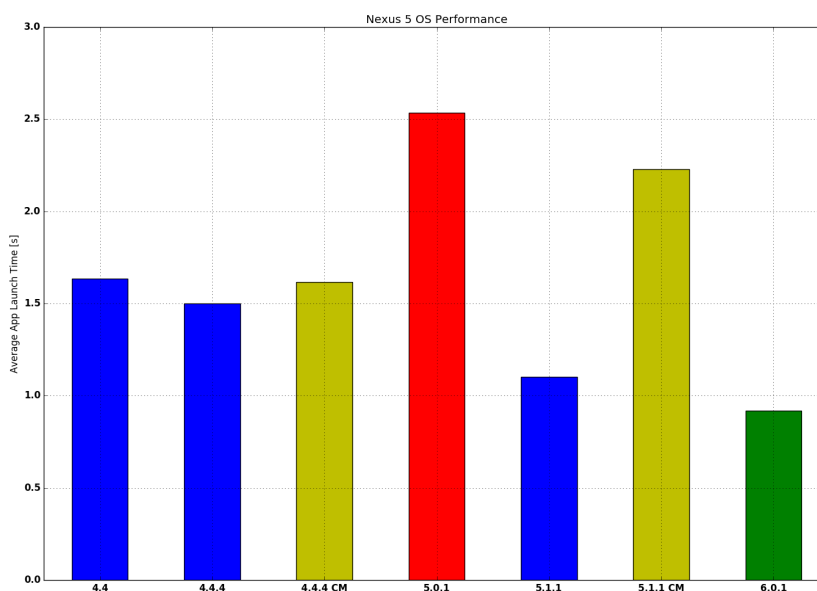


Figure 5.14: Performance of the Nexus 5 with different versions of Android. The 19 most popular apps among DiscoMark users were started for 10 runs, and this was repeated several times for each bar in the graph. Outliers were removed

## 5.8 Galaxy S6 Battery Saver Test

In this last experiment, we investigated the influence of screen brightness and the battery saver mode on the performance and battery life of the Samsung Galaxy

S6. The green bars denote that the battery saver was active, and the percentage on top of the bars shows how much battery was used up during the test. The height of the bars themselves shows the average launch-time for an app, i.e., the overall performance (higher means slower). The experiments were carried out for three different screen brightness settings: 0, 50 and 100%.

Figure 5.15 shows that the battery saver mode did in fact not save any battery, but decreased performance by roughly 50%. As expected, screen brightness makes a big difference. A brightness setting of 100% used about 50% more battery than when the brightness was set to 0%. The test was performed by selecting 20 apps and performing the benchmark for 30 runs. The duration of the entire benchmarking process was around 75 minutes.

From these results we can conclude that turning on battery saver is practically useless when the (Samsung) phone is being used actively, and strongly decreases performance. Battery saver modes likely prolong battery life when the phone is in standby, by limiting background processes, synchronization and network activity. So, until Samsung comes up with a smarter battery-saver, that does not reduce performance during active use, users would need to manually enable/disable the battery saver. We have not tested the behaviour of other manufacturer's battery saver modes, and therefore leave this for future work.

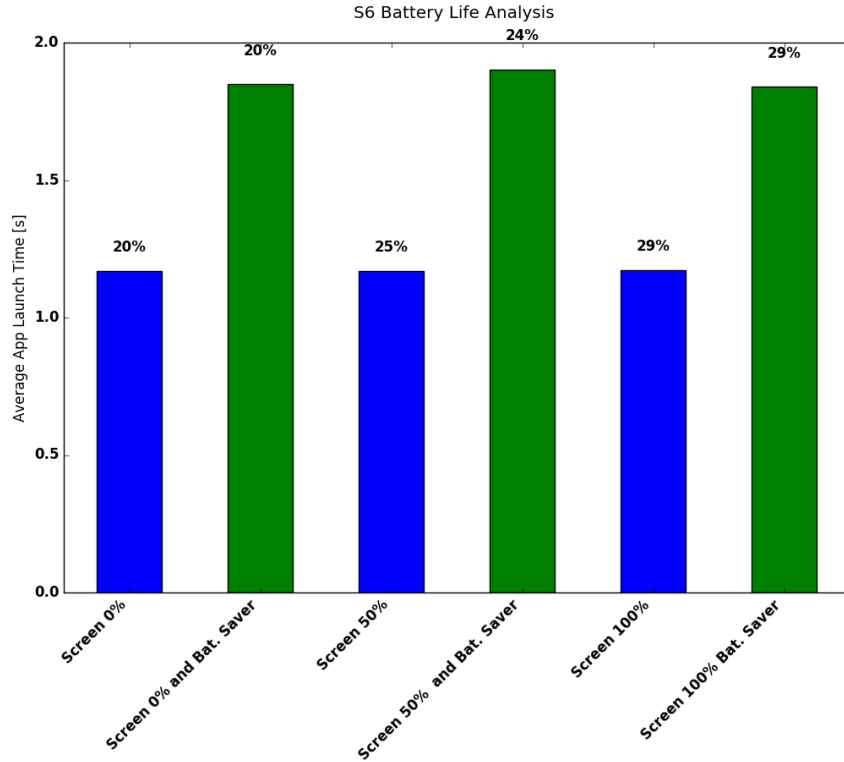


Figure 5.15: Influence of different screen-brightnesses and the battery-saver mode on the battery life and performance of the Samsung Galaxy S6. The percentages above the graphs denote how much battery-life was used up after the test. For the test, the 19 most popular apps among DiscoMark users were started for 30 runs, and this was then repeated several times for each bar in the graph. Outliers were removed.

## 5.9 Discussion of Results

Our experiments showed the performances of three phones under different circumstances. While this is interesting, the goal of these experiments was not just to compare these phones, but also to gain a better understanding of how well DiscoMark works, and what factors influence real-world performance, and how it relates to performance-measurements by conventional benchmarking applications. The insights gained from the experiment will be crucial in determining which steps to take next, and in improving DiscoMark itself and prepare it for release.

Below, we list the most important findings from our experiments:

- Real world performance depends on many variables:
  - Phone hardware
  - OS version
  - Number of installed Apps
  - Running background processes
  - Running updates in the background
  - Network connection speed/quality
  - Time since last restart
  - Having recently installed new apps without restarting
- Measuring only one app is not enough to predict a phones performance for all other apps
- Restarting helps to increase performance. In certain cases, the improvements are dramatic
- The performance of all phones decreased when installing more and more apps. The performance decreases ranged between 30%-70%
- Conventional benchmarks (by design) fail to represent real-world performance, and this also showed in our experiments. DiscoMark on the other hand allowed us to get quantitative insights that would not be possible otherwise. We therefore conclude that we are on the right track, and that DiscoMark is indeed able to solve a new problem.
- The Galaxy Note II's performance is closer to the other phones than benchmarks like AnTuTu would suggest, further proving that synthetic benchmarks are not an ideal indicator of real-world performance
- There are large performance differences between versions of Android on a Nexus 5. Overall, the performance has been increasing with newer versions of Android, with the exception of 5.0.1. Furthermore, the tested versions of CyanogenMod were inferior to their Vanilla counterparts
- The Battery Saver mode on the Galaxy S6 should not be enabled during active usage, since it reduces performance by roughly 50% and does not save any battery

We gained many new insights through these experiments. However, controlled lab-tests can only bring us so far. The Android ecosystem is very complex and fragmented and it is not enough to just test three phones and then draw conclusions about all other phones. Furthermore, it is not feasible to test a large number of phones in many different configurations and scenarios in controlled tests. Therefore, the next crucial step will be to use crowd-sourcing in

order to get real-world data from real users. To that end, we released DiscoMark on Google Play and successfully promoted it to gain users. The results of the user-data analysis are presented in Chapter [6](#)

# User-Data Analysis

---

After successfully promoting DiscoMark, we experienced a surge in user numbers and a steady stream of benchmarking results. In this chapter, we analyse the collected user-data to answer interesting questions and get new insights.

## 6.1 DiscoMark Promotion and Statistics

In this section we list statistics that describe the user-base and data that we have collected as of April 7th 2016.

**App downloads:** 9322

**App rating:** 4.51/5.0 from 107 ratings

**Distinct users in dataset:** 4876

**Distinct phone models:** 1202

**Distinct phone manufacturers:** 160

**Most popular phones :**

1. Samsung Galaxy S6 (various models): 187
2. LG Nexus 5: 177
3. OnePlus One: 141
4. Huawei Nexus 6P: 130
5. Motorola Nexus 6: 89

**Most tested applications :**

1. Chrome: 5157
2. Google Play Store: 3648

3. YouTube: 3359
4. WhatsApp: 3346
5. Gmail: 3313
6. Facebook: 2633

**Total number of benchmark runs:** 12'213

**Total number of app-results:** 126'728

**Average number of runs per app-result:** 3.3

**Total number of app launches:**  $3.3 \cdot 112'464 = 418'202$

**Distinct tested applications:** 8935

## 6.2 Rebooting Improves Performance

When we discussed the experiments in Chapter 5, we saw that rebooting a phone can dramatically improve its performance, especially after one has just installed a number of new apps. In this section we present the results from the user-data analysis regarding the influence of rebooting (i.e. *uptime*) on performance.

Qualitatively, it is obvious that indeed, rebooting generally increases performance significantly. Figure 6.1 shows the launch-times of Chrome for the Nexus 5, while Figure 6.2 shows the results over *all* apps for the Nexus 5. In both cases, we see a clear increase in performance after rebooting (i.e. *uptime* < 1h). On average, the performance increase due to rebooting, measured over all apps, for all our users with a Nexus 5, is roughly 30%. The Nexus 6 (Figure 6.3) and Nexus 6P (Figure 6.4) seem to profit even more from rebooting. Not only is the speedup roughly 50%, but also the variance is much smaller, indicating that there are fewer abnormally long app-launch times, leading to an overall smoother and more reliable user-experience. The results look very different for the Samsung Galaxy S6, where rebooting only brings a small performance-increase, as shown in Figure 6.5. Finally, Figure 6.6 is a plot covering all phones and all apps, and shows that without constraining the dataset at all, we still see better performance for freshly rebooted phones.



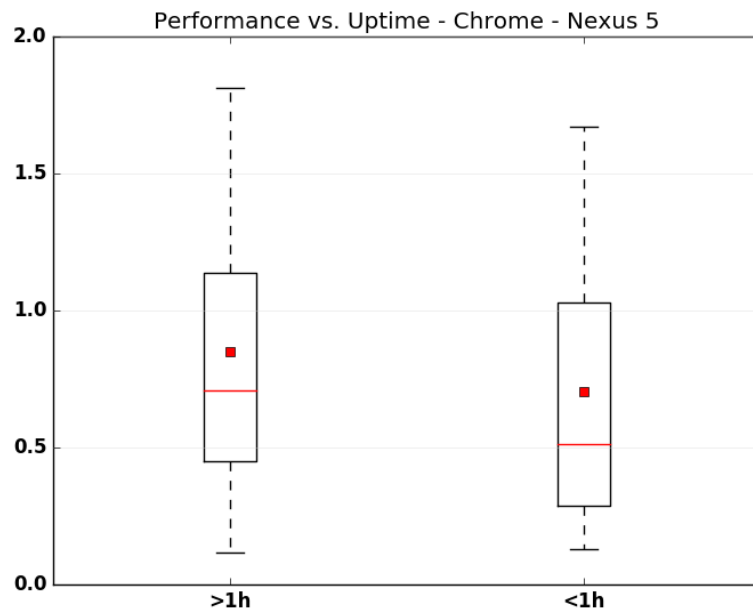


Figure 6.1: Effect of rebooting on the launch-times of Chrome on the Nexus 5

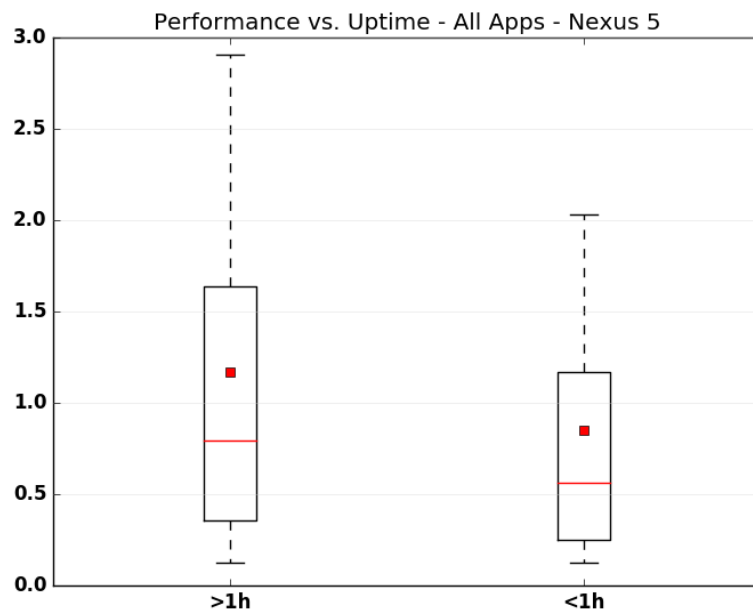


Figure 6.2: Effect of rebooting on the launch-times of all apps on the Nexus 5

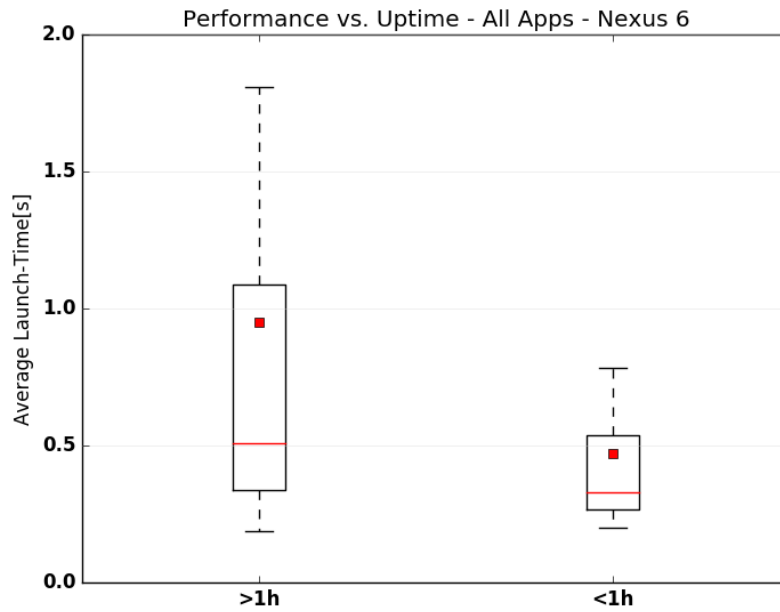


Figure 6.3: Effect of rebooting on the launch-times of all apps on the Nexus 6

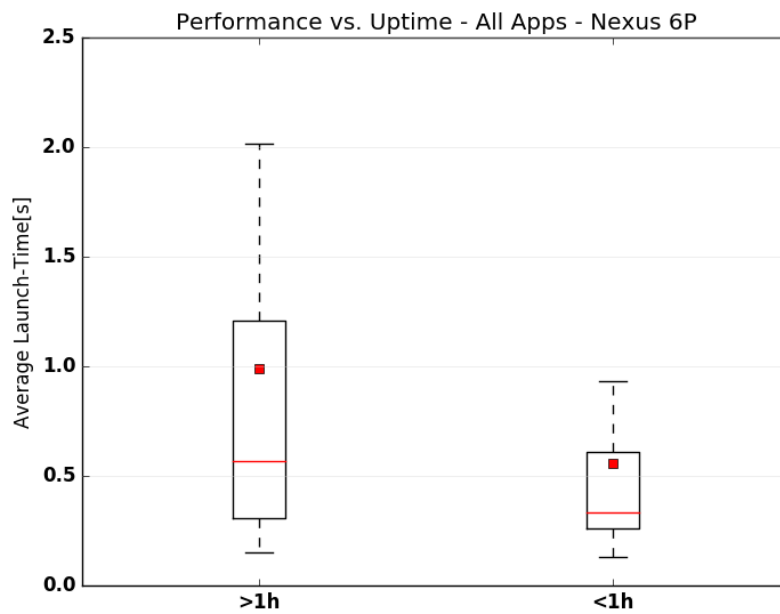


Figure 6.4: Effect of rebooting on the launch-times of all apps on the Nexus 6P

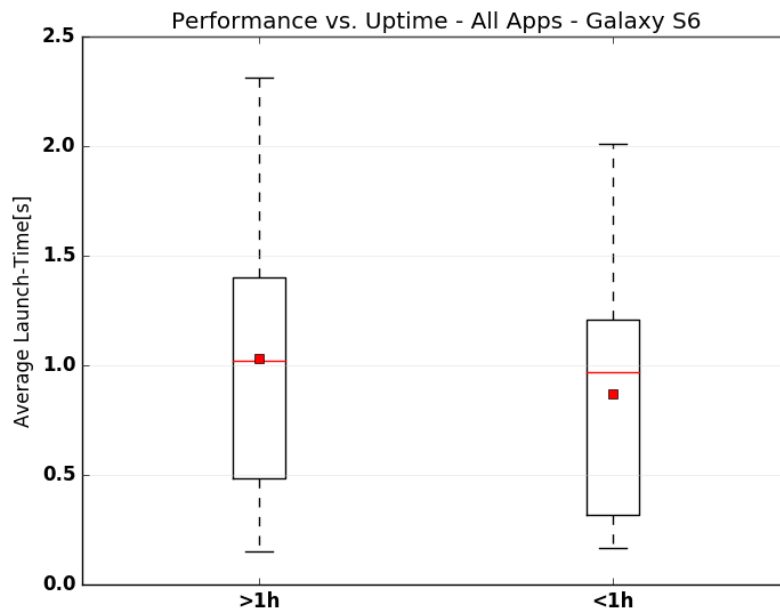


Figure 6.5: Effect of rebooting on the launch-times of all apps on the Galaxy S6

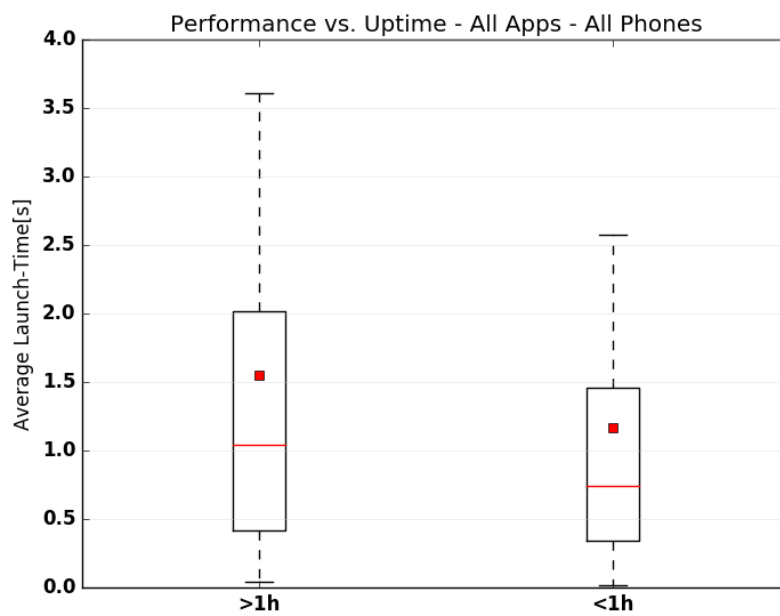


Figure 6.6: Effect of rebooting on the launch-times of all apps across all phones

### 6.3 Uninstalling Facebook and Messenger

In order to promote DiscoMark, we took advantage of a discussion around the Facebook Android app, and whether it slows down your phone. At that time, we performed a quick experiment on one phone to prove that FB indeed impairs your phone’s performance. We posted the results to Reddit and received an overwhelming reaction. Our post was Reddit’s #1 and received over 1200 comments, and the linked plots were opened more than 200’000 times. Newspapers like *The Guardian* and *Süddeutsche Zeitung* reported on it, as well as technology sites and blogs. Many users have since performed the exact same experiment themselves: They ran the test twice, once with and once without having FB installed. Therefore, we have a large dataset with little noise. Between those two tests, most parameters remained unchanged, most importantly, the number of installed apps and the number of tested apps. However, there is one important aspect that might very well be different between tests, and that is *uptime*. For example, a user first benchmarks his phone with FB installed, then uninstalls it, restarts his phone, and then does the benchmark again. As we have seen during the experiments from Chapter 5 and as was shown in Section 6.2, restarting a phone can increase the performance dramatically. In order to correct for this potential bias, we sliced the dataset by the *uptime*, where we define that a phone has been rebooted before a benchmark if it completes it within one hour after rebooting.

Figures 6.7, 6.8 and 6.9 show how the performance increases after uninstalling Facebook, sliced by different values for *uptime*. It is obvious that Facebook slows down the phones of our users, regardless of whether we correct for the uptime-bias. In line with our local, one-phone experiment from Section 6.1, the speedup is between 12 and 20%. We calculated the speedup as  $\frac{t_F - t_{noF}}{t_F} \cdot 100$ , where  $t_F$  denotes the average launch-time *with* Facebook installed, and  $t_{noF}$  stands for the average launch-time *without* Facebook installed. We also analysed what happens when one uninstalls Messenger, or both Messenger *and* Facebook, and found very similar results. It is noteworthy that it does not matter whether one has only one or both apps installed, the slow-down is the same. Therefore, in order to increase the performance of one’s phone, both apps need to be uninstalled.

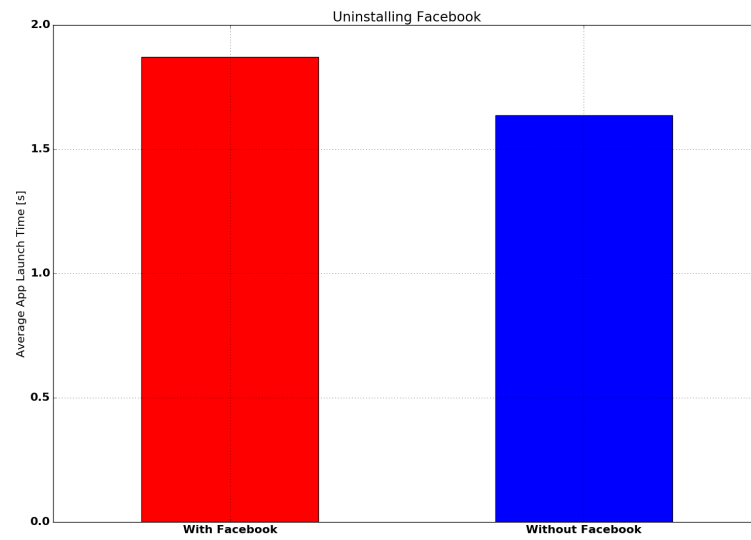


Figure 6.7: Performance with and without Facebook for  $uptime > 1h$ . Data from 158 distinct users.

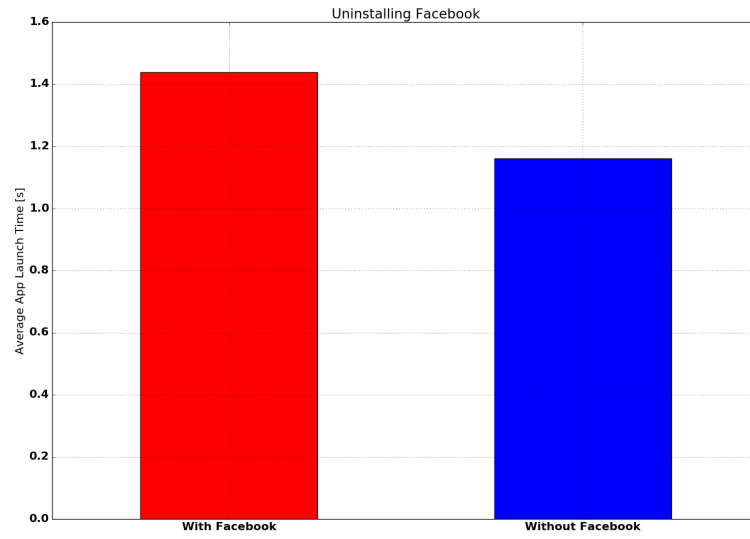


Figure 6.8: Performance with and without Facebook for  $uptime < 1h$ . Data from 129 distinct users.

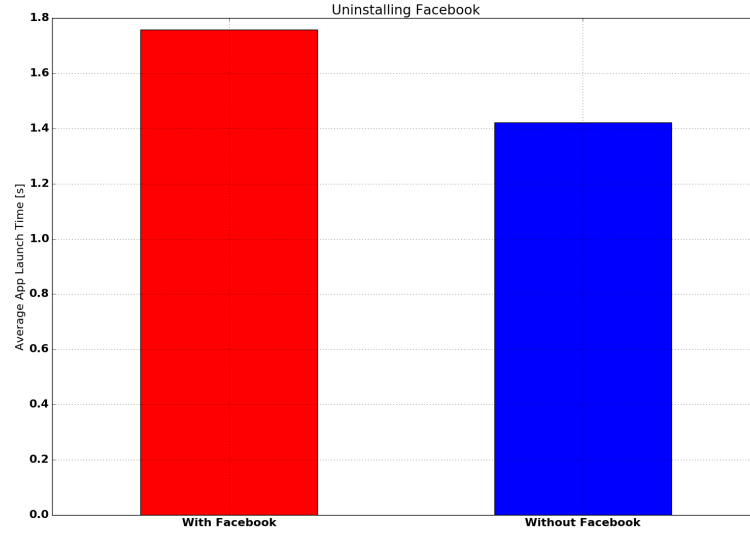


Figure 6.9: Performance with and without Facebook for all uptimes. Data from 370 distinct users.

## 6.4 Cold and Hot Starts

In this section we investigate behaviour of app-launch times with regard to caching. It is assumed that as long as there is space in memory, an opened app should remain cached and therefore, open faster next time. Concretely, this means that when using DiscoMark, the launch-times during the first run should be the slowest (assuming the apps have not already been in memory due to prior use), and then gradually get faster with the second and third run. However, as we have realized from lab-experiments, the actual multitasking implementations seem to differ strongly between different manufacturers. We will henceforth refer to caching/multitasking performance as MTP (MultiTasking Performance).

Below figures show how the launch-times behave for the first three runs for the Nexus 5. In Figure 6.10 we see what we would normally expect to happen when only a few apps are involved: Slow first run and then strong performance-increase for the subsequent runs. MTP gradually decreases until 20-25 apps are reached (i.e. users picked 20-25 apps to be included in the benchmark), where there is no performance increase anymore. Above that, we even see a decrease in MTP, as seen in Figure 6.14. This is likely due to background-activities being

started by the opened applications that then start requesting the CPU.

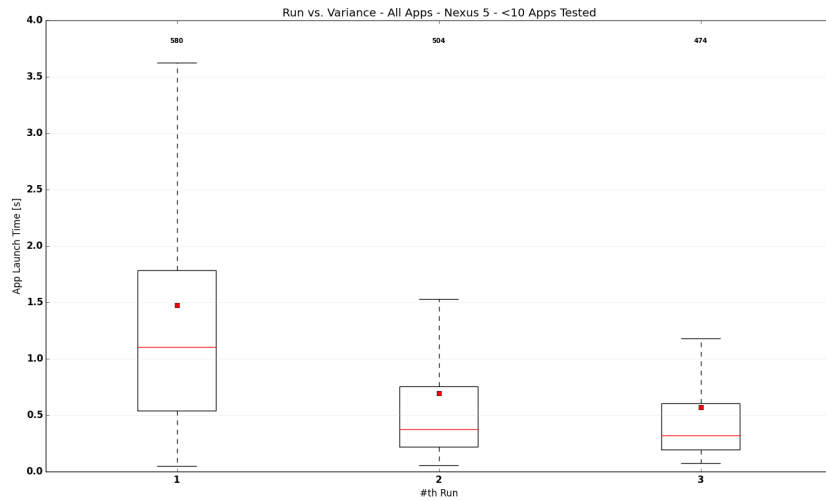


Figure 6.10: Evolution of launch-times and variance with the number of runs for the Nexus 5 with fewer than 10 tested apps.

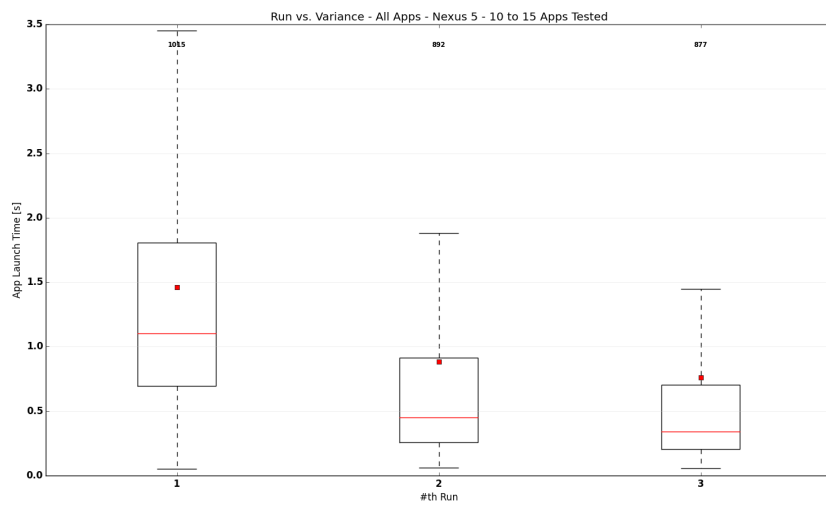


Figure 6.11: Evolution of launch-times and variance with the number of runs for the Nexus 5 with 10-15 tested apps.



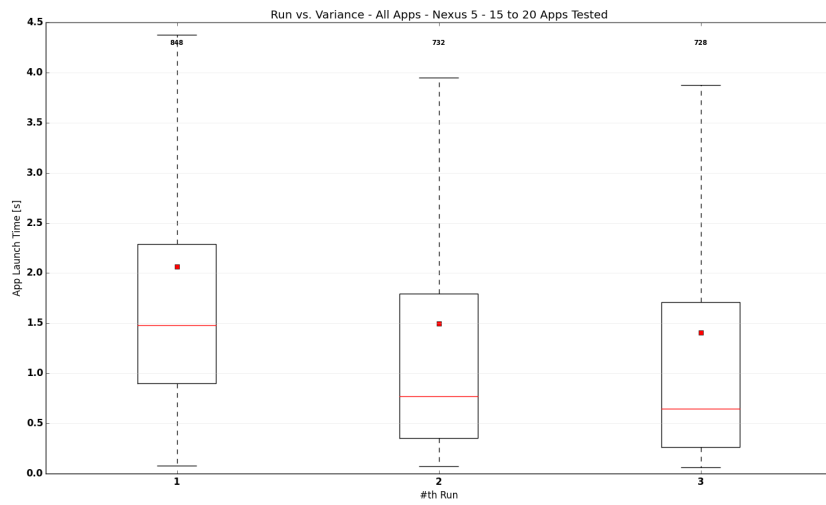


Figure 6.12: Evolution of launch-times and variance with the number of runs for the Nexus 5 with 15-20 tested apps.

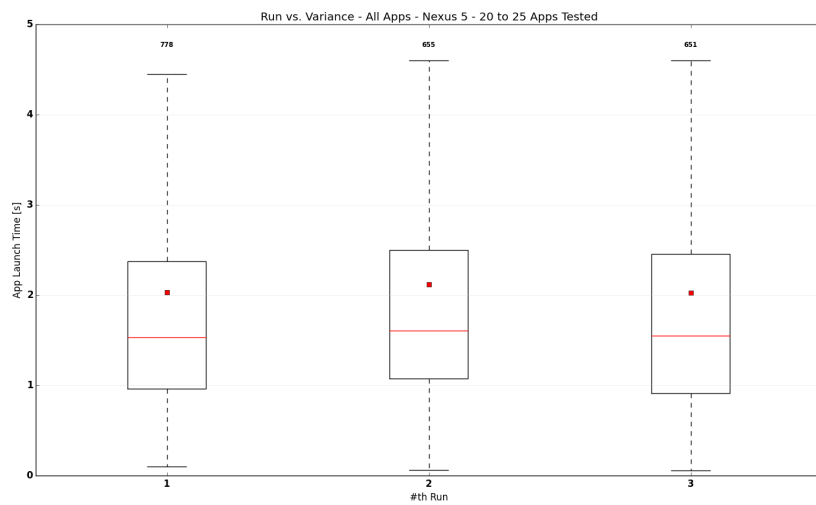


Figure 6.13: Evolution of launch-times and variance with the number of runs for the Nexus 5 with 20-25 tested apps.

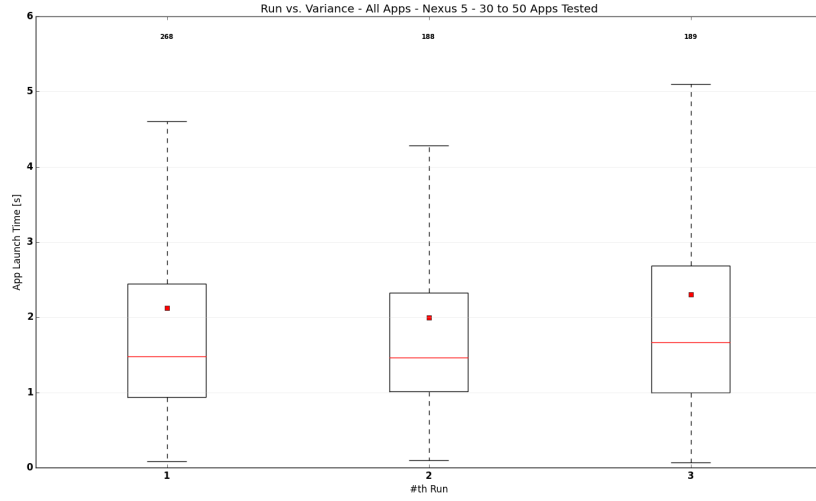


Figure 6.14: Evolution of launch-times and variance with the number of runs for the Nexus 5 with 30-50 tested apps.

Now we move on to the newer Nexus 6, which not only has a faster CPU, but also 3GB RAM instead of the Nexus 5's 2GB. We expect that the Nexus 6 can go higher than 20-25 tested apps due to that additional 1GB of memory. Figure 6.15 again shows the expected behaviour, where MTP increases rapidly after the first run. In order to save space, and since the following plots are qualitatively the same as for the Nexus 5, we jump directly to 20-25 tested apps as shown in Figure 6.16. Surprisingly, the Nexus 6's MTP “breaks down” at the same time as it did for the Nexus 5, despite its larger memory.

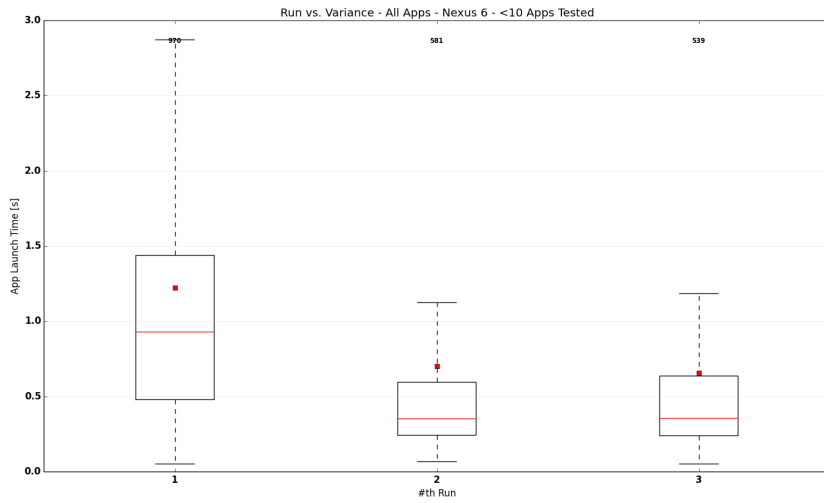


Figure 6.15: Evolution of launch-times and variance with the number of runs for the Nexus 6 with fewer than 10 tested apps.

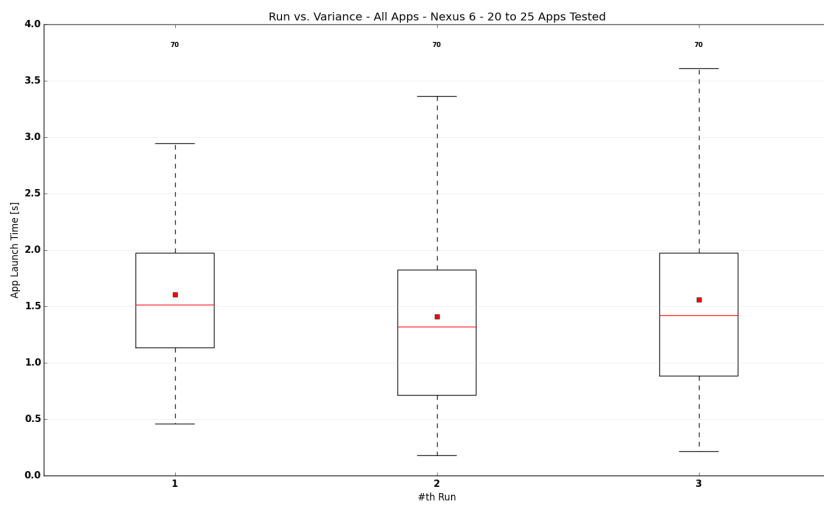


Figure 6.16: Evolution of launch-times and variance with the number of runs for the Nexus 6 with 20-25 tested apps.

In the following, we will look at the MTP of the OnePlus One. The One is a special phone, because it is shipped with Cyanogen OS.<sup>1</sup> It is obvious that

<sup>1</sup><https://cyngn.com/get-cyanogen-os>

while qualitatively similar to the plots of the Nexus 5 and 6, the One's performance increases much more after the first run, leading us to think that the multitasking/caching implementation of Cyanogen OS is very good, and exceeds that of stock Android. This extreme performance will also be visible later on when we compare the real-world performance of different phones (Section 6.6). However, as before with the Nexus 5 and 6, once we reach 20-25 tested apps, the caching-benefit vanishes.

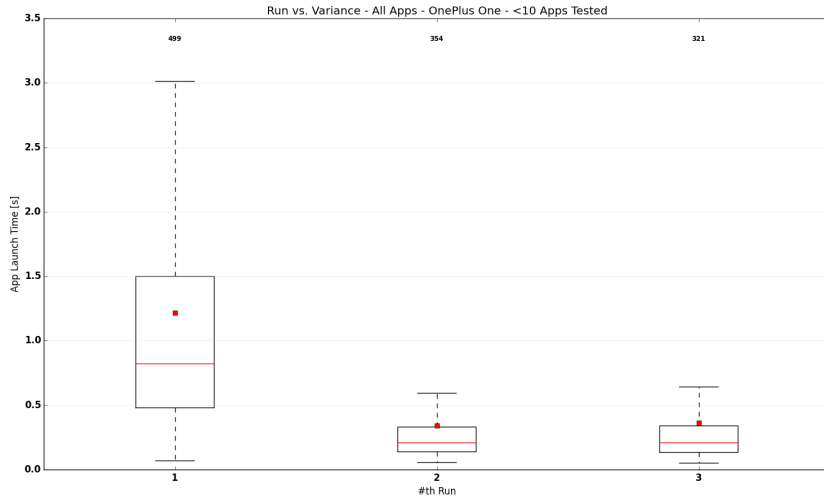


Figure 6.17: Evolution of launch-times and variance with the number of runs for the OnePlus One with fewer than 10 tested apps.

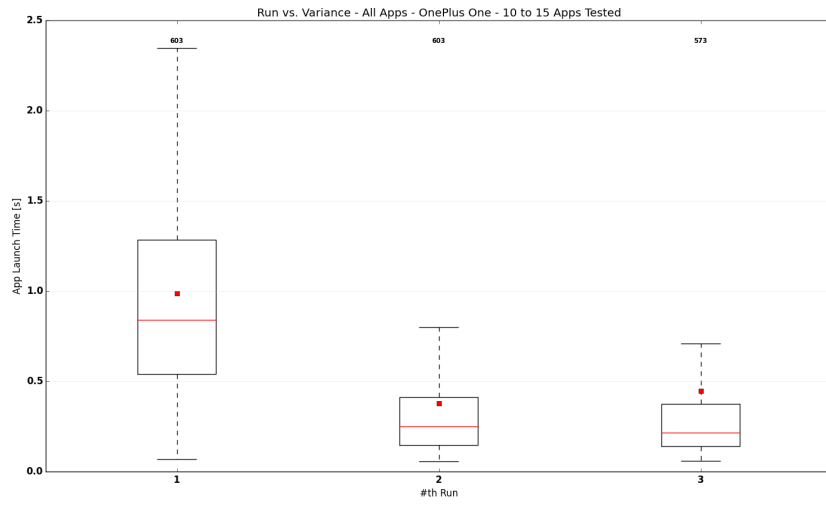


Figure 6.18: Evolution of launch-times and variance with the number of runs for the OnePlus One with 10-15 tested apps.

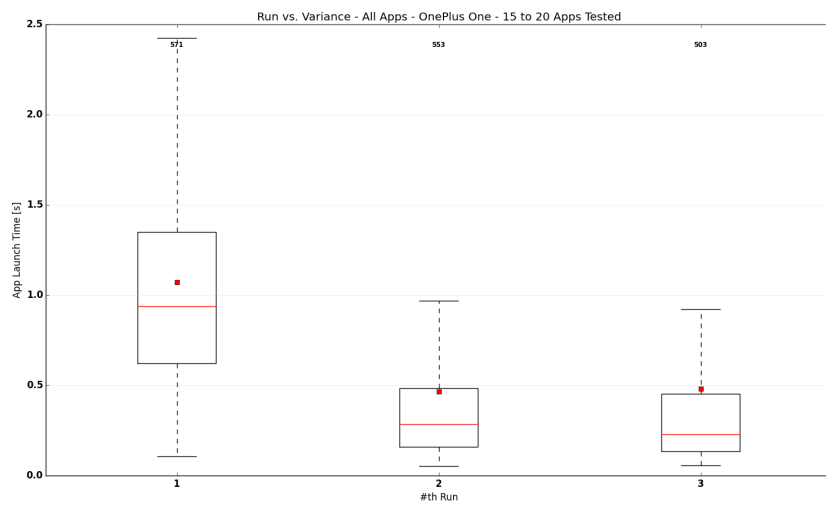


Figure 6.19: Evolution of launch-times and variance with the number of runs for the OnePlus One with 15-20 tested apps.

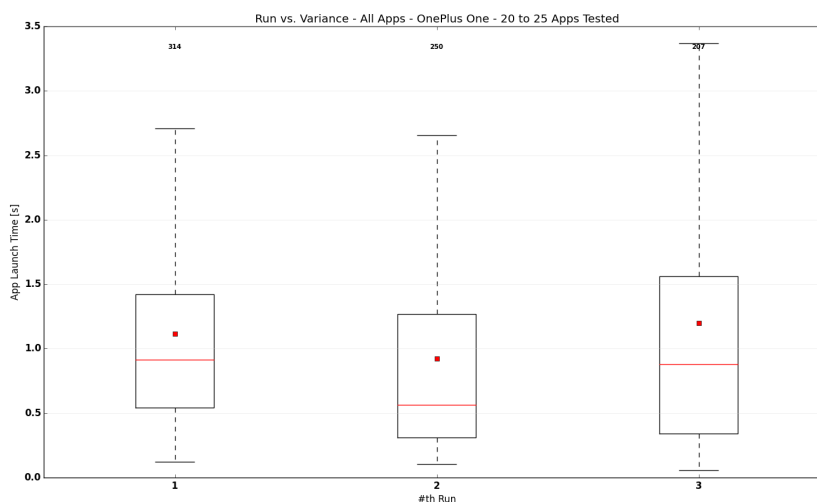


Figure 6.20: Evolution of launch-times and variance with the number of runs for the OnePlus One with 20-25 tested apps.

Another odd case is the Samsung Galaxy S6. While equipped with a very capable CPU, which allows it to outperform many rivals, we think that it could actually be much faster if it had better MTP. As can be seen in Figure 6.21, for fewer than ten tested apps, MTP is decent and there is nice increase in performance after the first run. However, the performance-increase is less than for the Nexus 5 and 6, and nowhere near that of the OnePlus One. When we look at the results for 10-15 tested apps in Figure 6.22, the caching benefit is already completely gone. The good thing is that no matter how many apps are tested (as long as it is more than 10), the performance stays the same, as is shown in Figure 6.23. These results were confirmed in controlled lab experiments as well, where the Galaxy S6 was tested against the Nexus 5X and Nexus 6.

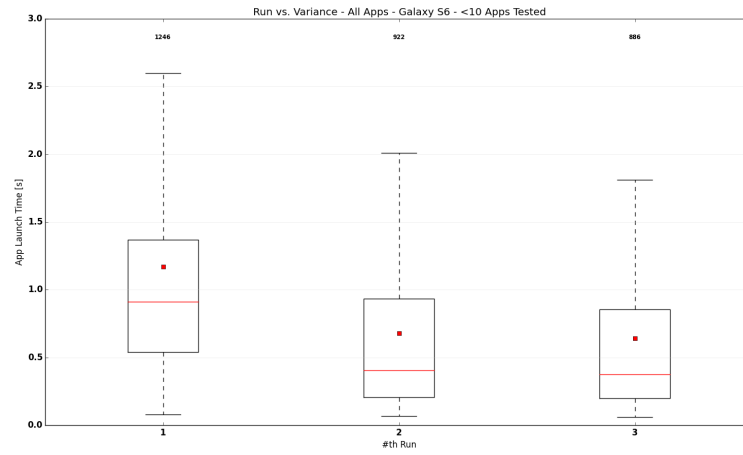


Figure 6.21: Evolution of launch-times and variance with the number of runs for the Galaxy S6 with fewer than 10 tested apps.

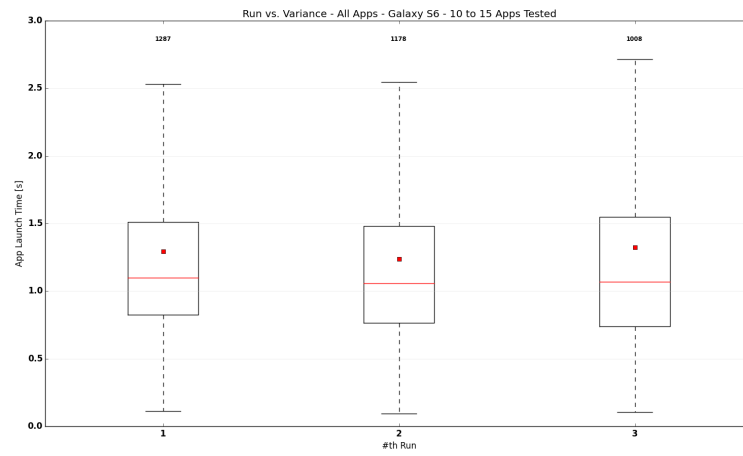


Figure 6.22: Evolution of launch-times and variance with the number of runs for the Galaxy S6 with 10-15 tested apps.

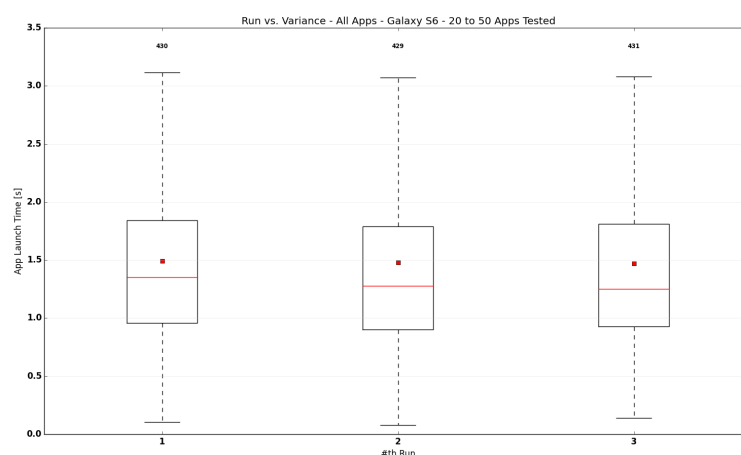


Figure 6.23: Evolution of launch-times and variance with the number of runs for the Galaxy S6 with 20-50 tested apps.

To summarize this section, we can draw following conclusions:

1. There is no difference between 2GB and 3GB of RAM when it comes to MTP
2. There are significant differences between manufacturers when it comes to MTP
3. For many phones there seems to be a maximum of 20-25 open apps that one should not exceed in order to still benefit from caching. However, fewer seems to be better, and therefore, frequently clearing unused apps, or rebooting the phone, is recommended.
4. For the Samsung Galaxy S6 (and likely other Samsung phones), the limit is as low as ten apps. Luckily, the S6's fast hardware allows it to perform well, despite its bad MTP.

## 6.5 # Installed Apps vs. Performance

In Chapter 5 we presented results from controlled lab tests, where we investigated the influence of the number of installed apps on a phone's real-world performance. In this section, we evaluate the data of our users to get a better idea of what is happening "in the wild". We only look at two phones; the Nexus 5 and Nexus 6. The results of all other popular phones fall in line with the ones presented below.



Figure 6.24 shows the launch-times of Chrome plotted against the number of installed apps. The decrease in performance is obvious, and confirms the lab tests we performed in 5. Once we extend the dataset to include not just Chrome, but all apps, we get a very similar result. The Nexus 6, as seen in Figure 6.26, exhibits the same qualitative characteristics. For this kind of analysis, our dataset is not large enough to draw quantitative conclusions. However, the qualitative trend is clear, and it is therefore recommended to keep one's phone as clean as possible, and uninstall unused apps.

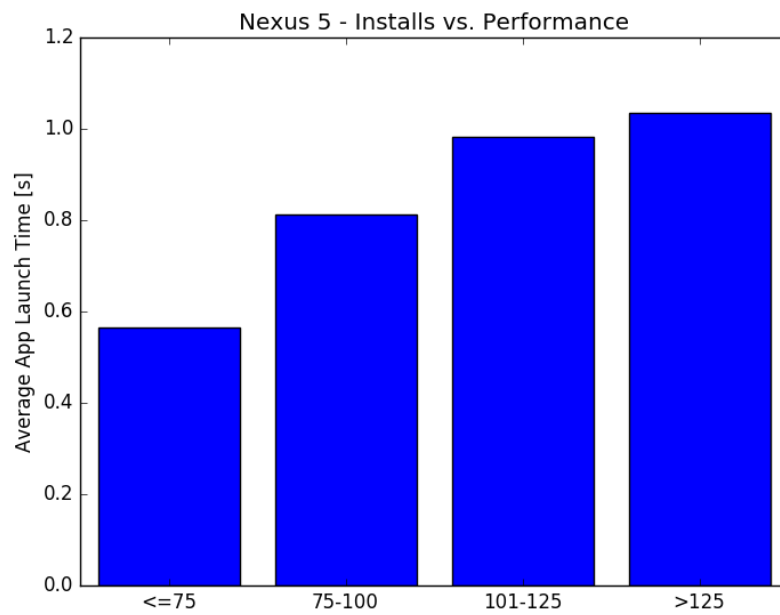


Figure 6.24: Performance (launch-time of Chrome) decreases strongly with increasing number of installed apps for the Nexus 5.

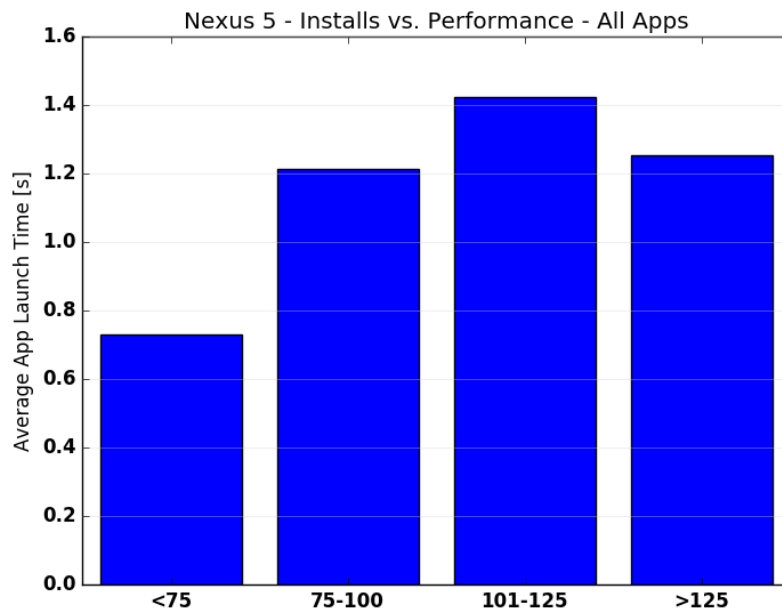


Figure 6.25: Performance decreases strongly with increasing number of installed apps for the Nexus 5.

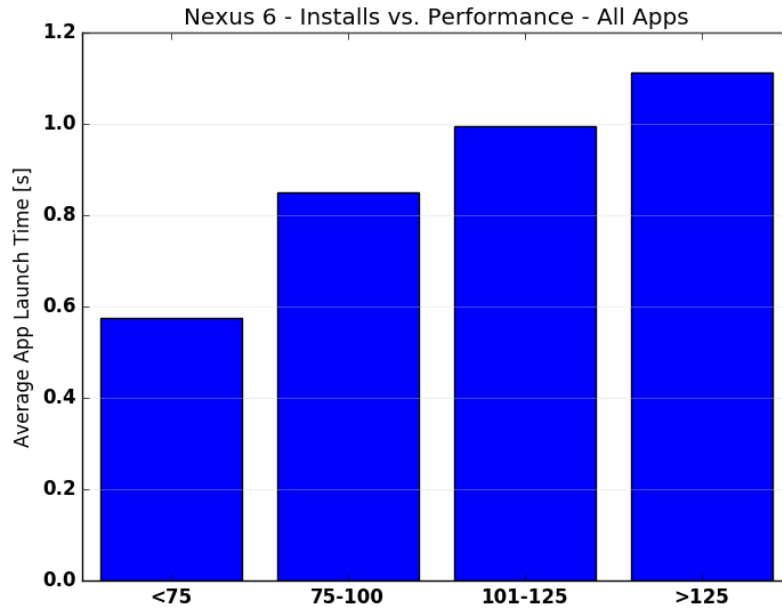


Figure 6.26: Performance decreases strongly with increasing number of installed apps for the Nexus 6.

## 6.6 DiscoMark vs. Geekbench

One of the main goals of this thesis was to investigate how well synthetic benchmarks represent real-world performance. To that end, we plotted the most popular phones' DiscoMark performance vs. their Geekbench Multicore CPU performance. For a phone to be included in the graph, it must be represented by at least 30 distinct users in our dataset. We sliced the dataset by the run-number, i.e., we created a separate plot for cold-starts (1st run) and hot-starts (subsequent runs). We expect that the cold-start performance will mostly depend on the CPU-power of the phones, where the hot-start performance will be strongly influenced by multitasking/caching-performance, as we have discussed in Section 6.4. Both axes were normalized, where 100 was assigned to the best-performing phone. A phone with score 80 performs twice as fast as one with score 40.

Figure 6.27 shows the resulting scatter plot when we only consider the first run of each benchmark, i.e., cold starts, without the benefit of caching. We can see that overall the increase in real-world performance is linear with increasing CPU-performance. There are a few outliers that are performing better than their CPU-score suggests, e.g., the Moto G3 ( $\geq 1\text{GB}$ ). Most notably however are the Galaxy Note 5 and the OnePlus One, which are both performing significantly

better than the rest. While this could have been expected for the Note 5, being the newest phone in the plot, it is certainly a surprise for the OnePlus One. Compared to its peers with similar (Z3 Compact, LG G2, etc.) or better CPUs, it performs up to twice as fast.

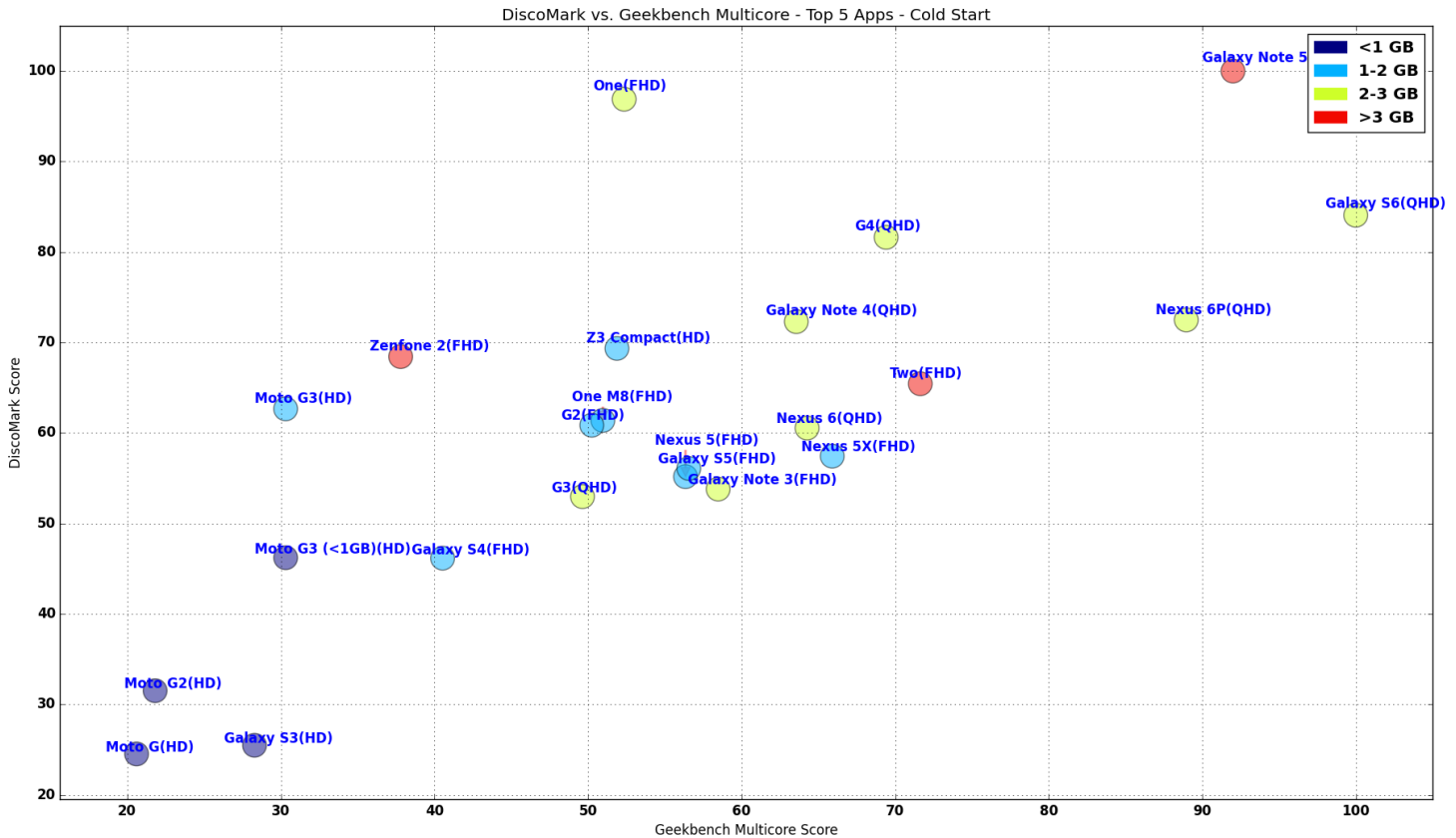


Figure 6.27: Real-world performance as measured by DiscoMark plotted against the multi-core CPU scores of Geekbench. Only the first run of each test is considered (cold start)

Keeping in mind the results we showed in Section 6.4, where we saw that the OnePlus One had incredible MTP, we expect it to perform even better when we consider hot-starts only. Furthermore, we saw that the Samsung Galaxy S6 had poor MTP, and we speculated that this should be the case for other Samsung phones as well, which is why we expect them to lose ground against other phones, such as the Nexus-family. Indeed, Figure 6.28 confirms our hypothesis. The OnePlus One is even more dominant, now that it can profit from its strong

MTP. Additionally, most Samsung phones fell behind their direct competitors. For example, the Galaxy S6 and Note 5 were overtaken by the Nexus 6P, and the Galaxy Note 4 fell behind the Nexus 6. An interesting observation is that the Nexus 5X seems to perform poorly compared to its Nexus-siblings. In fact, Google recently released a software update for the Nexus 5X that should solve performance problems that have been bothering it. Therefore, in future, the Nexus 5X will likely perform better. Another phone that has much better real-world performance than its CPU-score would suggest is the Asus Zenfone 2, one of only a few phones using an Intel CPU. It outperforms most other phones that have better theoretical performance.

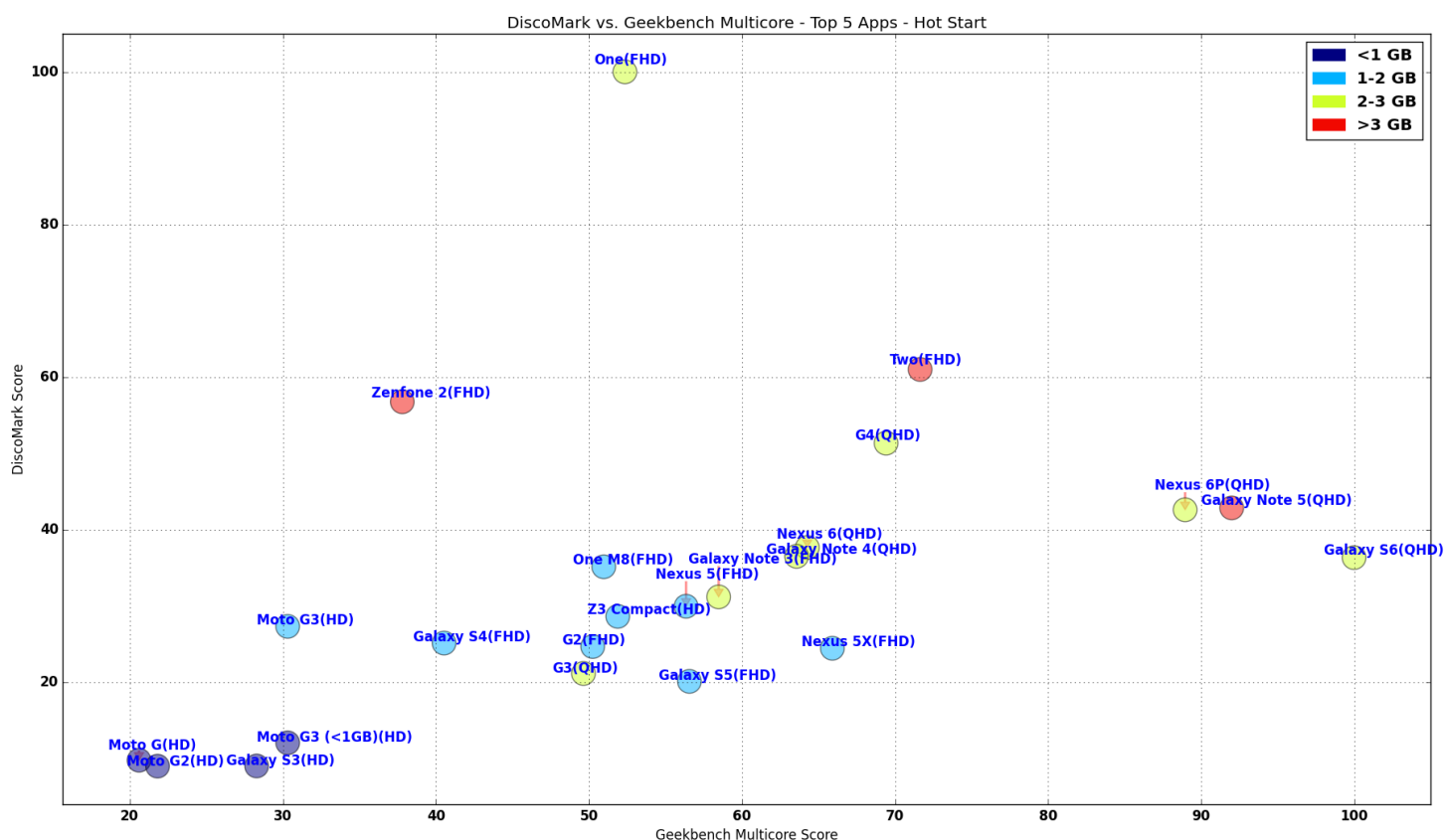


Figure 6.28: Real-world performance as measured by DiscoMark plotted against the multi-core CPU scores of Geekbench. The first run is ignored (hot start)

Finally, we remove the OnePlus One from Figure 6.28 to get Figure 6.29, so that the differences between the other phones can be seen more clearly. We

can now see the distinction between phones with less than 1GB of RAM and the rest. This is most clearly shown by the Moto G3, which has versions with different amounts of memory, but otherwise identical hardware. This finalizes our conclusion that a new phone should have 2GB of RAM. Having more than 2GB does not provide any benefits as of now, and having less than 1GB clearly impairs performance.

Furthermore, looking at all three plots together, it is clear that synthetic benchmarks merely provide a rough idea of how a phone might perform during real world usage. Stronger hardware certainly makes it more likely that a phone will perform better than one with older internals, but it is far from being a guarantee. By looking at the scatter plots one can find many such examples. One example is the vertical 60-70 band in Figure 6.29, where we have four phones with very similar hardware, but wildly varying DiscoMark scores.

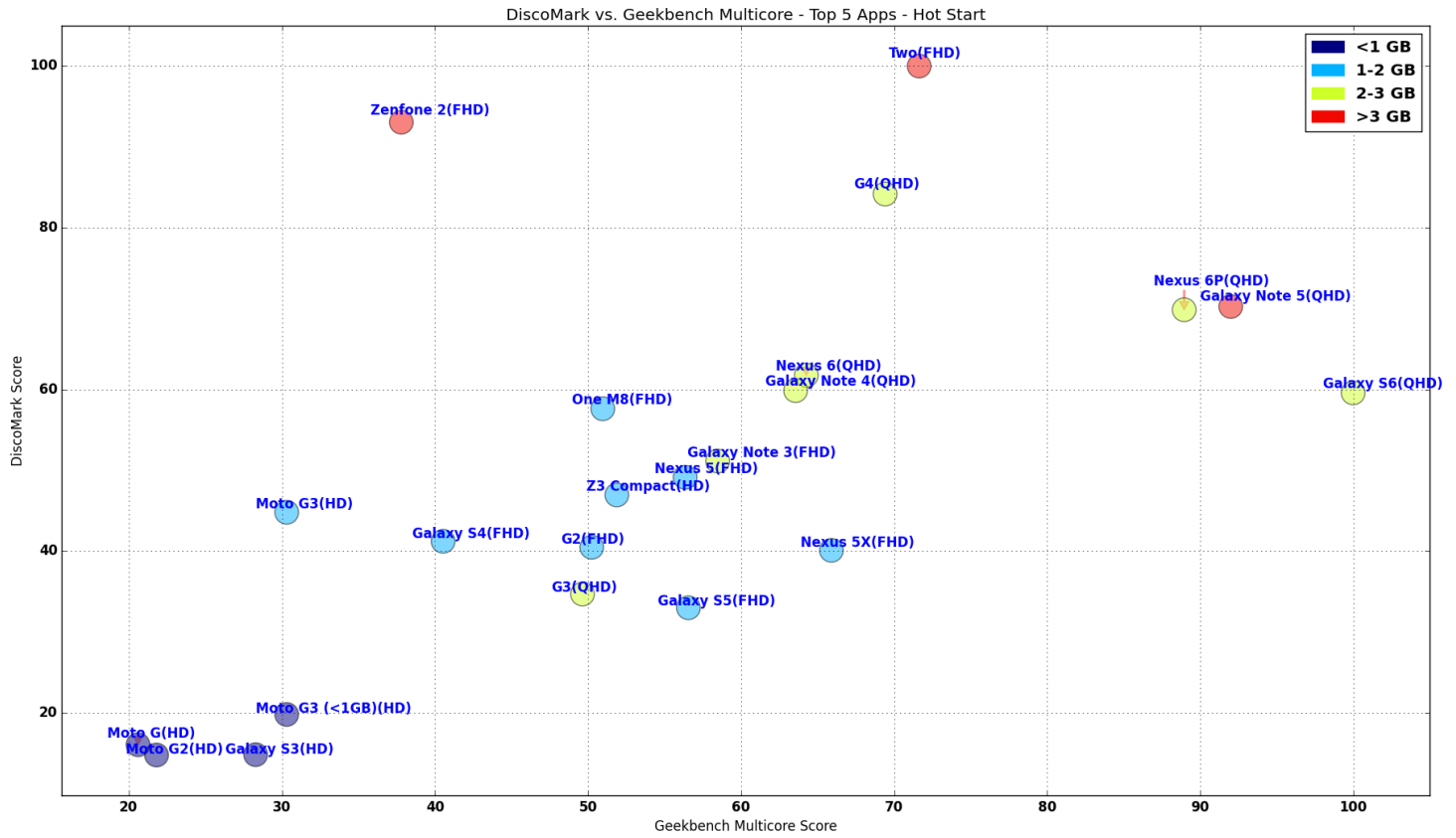


Figure 6.29: Real-world performance as measured by DiscoMark plotted against the multi-core CPU scores of Geekbench. The first run is ignored (hot start)

# Conclusion and Outlook

---

## 7.1 Conclusion

In this thesis, we laid much of the ground work necessary to investigate aspects of Android's performance from a real-world performance perspective. Among other things, we gained many insights into what influences performance, how to most accurately measure it, and what can be done to improve it. Furthermore, we showed that synthetic benchmarks, while useful to show qualitative trends, are not the end of the story, and do not accurately reflect real-world performance. Also, we saw that different manufacturers seem to have strongly different multitasking/caching implementations. Nonetheless, there are still unanswered questions, and many new ones came up during the course of this thesis. The Android platform is vast, complex and highly fragmented, and there are many interesting questions. In the following, we address the ones we find most worth investigating.

## 7.2 OS Performance Tests

In Section 5.7, we showed the performance of different versions of Android (stock and CyanogenMod) on a Nexus 5. While this was very insightful, it is far from the entire story. As mentioned above, the Android platform is highly fragmented and there are significant differences between phones, especially from different manufacturers. So, while CyanogenMod performs worse than Vanilla Android on a Nexus 5, it might very well be the other way around on a phone that is running a flavoured version of Android, such as Samsung TouchWiz or HTC Sense. Since these kinds of tests highly benefit from noise-free and accurate data, we believe that performing lab-experiments is the better way to go, as opposed to waiting for enough user-data to arrive. However, it needs to be investigated which phones are most suitable for this task, since not all are equally easy to be flashed with different versions of a stock ROM or custom ROMs, or might even lack support completely from, e.g., CyanogenMod. This leads us to another



interesting point; CyanogenMod itself has different flavours, and it would be interesting to compare them to each other.

### 7.3 DiscoMark and User-Data

The amount of data we gathered from our users is certainly respectable, and allowed us to draw many interesting conclusions and put them into quantitative terms, where before one could only guess. However, there are still many unanswered questions, and answering them would unfortunately require a vastly larger dataset. To that end, DiscoMark needs to be refined to attract many more users. As of now, DiscoMark is capable, but not easy to use and understand. The users have to be willing to put some work in to get valuable information for themselves. For example, DiscoMark could be extended into a real-world performance benchmark framework, where the user does not need to do anything except press a button, and the app then performs a series of different tests, with different apps selected, different number of runs, different number of apps running in the background, and different settings, etc. Through that, one could analyse a phone's performance in a more detailed and structured way, e.g., caching performance, performance when a few/many apps are used, how much rebooting affects performance, etc. Furthermore, it should be made easier for users to get the desired information in an understandable fashion. As of now, they can export the results in a CSV file and do their own analysis, however, this could be automated and statistics and graphs could be displayed inside the app. Much of the analysis that was performed to create the graphs in this report could be integrated into DiscoMarks backend.

The improvement of DiscoMark's user-friendliness would of course serve the purpose of attracting many, many more users. We are certain that incremental increases in user-number and dataset-size are not useful anymore, but an exponential increase is what would be needed. As of now, DiscoMark has nearly 10'000 downloads, but we expect to need at least 100'000, or better more than a million, downloads, in order to increase our dataset to a point where we are not limited by sample-sizes anymore. During the analysis of the user-data, we could often not restrict the SQL queries as much as we wished, and that was for our most popular phones. For example, we might only want to look at a certain phone with uptime less than an hour, between 100 and 120 installed apps, for benchmarks with between 5 and 10 runs and 15 to 20 tested apps, and so on. This is just an illustration that shows how fast a seemingly large dataset can become inadequate when one starts asking detailed questions. Another advantage of having a much larger dataset, and thus being able to restrict it more strongly, is the ability to make much more accurate, quantitative observations. As of now, we are rarely confident to make more than qualitative statements, and had to revert to lab-experiments for that (exception is the uninstalling Face-

book experiment from Section 6.3, where we had a highly cohesive and focused dataset).

## 7.4 DiscoMark as a Service

The fact that DiscoMark is implementing Android’s `AccessibilityService` interface is both blessing and curse. Once a user sees that DiscoMark is always active as a service in the background, he or she might decide to uninstall the app for fearing it might be another resource-hog. However, we could also take advantage having an ever-running service at our disposal, by collecting information in the background, even when the benchmark is not running. For example, we could monitor power consumption and voltages of the CPU, battery usage and keep track of which apps are running in the background. From this information, we could figure out if the user has resource/battery-hogs installed, and could prompt him/her to uninstall certain apps, and then perform another benchmark in order to detect if the change brought a performance-improvement. Furthermore, we could keep track of all apps the user opens by inspecting accessibility events, and thus extract his/her user-pattern. Knowledge of the user-pattern could be used to perform the benchmark in a more automated manner, and be useful to develop DiscoMark into a fully-integrated real-world benchmark framework, as discussed in Section 7.3.

# Bibliography

- [1] Google: Google developer, androidjunitrunner. <http://developer.android.com/reference/android/support/test/runner/AndroidJUnitRunner.html> Accessed: 2016-03-23.
- [2] Google: Google developer, uiautomator. <http://developer.android.com/training/testing/ui-testing/uiautomator-testing.html> Accessed: 2016-03-23.
- [3] Google: Google developer, espresso. <http://developer.android.com/training/testing/ui-testing/espresso-testing.html> Accessed: 2016-03-23.
- [4] Google: Google developer, ui/application exerciser monkey. <http://developer.android.com/tools/help/monkey.html> Accessed: 2016-03-23.
- [5] Google: Google developer, monkeyrunner. [http://developer.android.com/tools/help/monkeyrunner\\_concepts.html](http://developer.android.com/tools/help/monkeyrunner_concepts.html) Accessed: 2016-03-23.
- [6] Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: Timing- and touch-sensitive record and replay for android. In: Software Engineering (ICSE), 2013 35th International Conference on. (May 2013) 72–81
- [7] Zhu, M.H.Y., Peri, R., Reddi, V.J.: Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem
- [8] AnTuTu: Google play entry for antutu. <https://play.google.com/store/apps/details?id=com.antutu.ABenchMark> Accessed: 2015-12-30.
- [9] Ltd., K.: Google play entry for gfbench. <https://play.google.com/store/apps/details?id=com.glbenchmark.glbenchmark27> Accessed: 2015-12-30.
- [10] Oy, F.: Google play entry for 3dmark. <https://play.google.com/store/apps/details?id=com.futuremark.dmandroid.application> Accessed: 2015-12-30.
- [11] Kim, J.M., Kim, J.S.: Androbench: Benchmarking the storage performance of android-based mobile devices. In Sambath, S., Zhu, E., eds.: Frontiers in Computer Education. Volume 133 of Advances in Intelligent and Soft Computing. Springer Berlin Heidelberg (2012) 667–674

- [12] Qualcomm Innovation Center, I.: Google play entry for vellamo. <https://play.google.com/store/apps/details?id=com.quicinc.vellamo> Accessed: 2015-12-30.
- [13] Pandiyan, D., Lee, S.Y., Wu, C.J.: Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - mobilebench. In: Workload Characterization (IISWC), 2013 IEEE International Symposium on. (Sept 2013) 133–142
- [14] Hu, Y., Azim, T., Neamtiu, I.: Versatile yet lightweight record-and-replay for android. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015, New York, NY, USA, ACM (2015) 349–366
- [15] Google: Google developer, accessibilityservice. <http://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html>
- [16] Google: Google developer, accessibilitynodeinfo. <http://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo.html> Accessed: 2015-12-30.