Evaluating the Performance of Android Layout Frameworks: A Case Study

Wannes Ghielens
University of Antwerp, Belgium

Gulsher Laghari
University of Antwerp, Belgium
Universirty of Sindh, Pakistan

Serge Demeyer
University of Antwerp, Belgium
Flanders Make, Belgium

Abstract—**The ANDROID LAYOUT FRAMEWORK has a number of performance pitfalls that have been documented over the years in various sources. To overcome these issues, several alternative layout frameworks have emerged that claim a performance gain. Unfortunately, these performance gains are never validated in practice. In this paper, we compare the performance of the four most popular alternatives (namely, DATA BINDING, ANKO, CONSTRAINTLAYOUT, and LITHO) to ANDROID LAYOUT FRAMEWORK by means of three different scenarios. We conclude that the frameworks vary a lot in their performance. Nevertheless, there is no single best solution suggesting that hybrid combinations are worthwhile to investigate.**

I. Introduction

With the rapid increase in the smartphone market, the mobile apps market is simultaneously increasing at massive scale. As of March 2017, the leading app stores Google Play and Apple's App Store had 2.8 million and 2.2 million apps available for download respectively [12].

Despite this massive growth, the quality of the apps varies substantially suffering from various issues [10]. Large scale studies of both Android and iOS apps report that the majority of apps suffer from performance issues [14]. Liu et al. found that 75.7% of performance bugs are GUI lagging, which can significantly reduce responsiveness of the app [16]. Similarly, a poor and complex UI layout may severely slow down the app, rendering the app unresponsive—one of the major issues for negative user reviews [14]. The main causes for these performance issues are two-fold: (i) complex layout hierarchies and (ii) runtime *layout inflation*.

(i) Complex layout hierarchies are a major bottleneck for app performance. For instance, a LinearLayout widget can be used to arrange widgets in a horizontal or a vertical fashion. These widgets can be combined as simple building blocks for more complex arrangements. The result is a tree of widgets each having a horizontal or vertical arrangement. There are however far more expressive widgets than LinearLayout that could achieve the same end result with just a single widget or a significantly lower amount. Unfortunately, these widgets are more complicated to use and therefore less popular amongst new developers. In general, reducing layout hierarchy complexity actually increases the complexity of the layout definition, and makes it harder to reason about for both new and trained developers.

(ii) Runtime layout inflation is a consequence of specifying the GUI layout of an app with an XML structure parsed at runtime on the device. This parsing step is named *layout inflation* and is often time-consuming, in particular because some XML structures exploit reflection. Indeed, each node inside the XML represents a *view object* in the layout tree and every attribute represents a property on the *view object*. Inflating these *view objects* and setting their properties usually happens through reflection and reducing this step would greatly improve performance [9], [21], [22].

Consequently, the Android community has forwarded guidelines to avoid common layout performance pitfalls [8]. On top of that, several alternative layout frameworks have emerged that either abstract away the flattened hierarchy or simply reduce the performance impact of nesting layouts. All of these alternatives claim a performance gain, but these performance gains have never been validated in practice. In this paper, we compare the performance of four alternatives (namely, DATA BINDING, ANKO, CONSTRAINTLAYOUT, and LITHO) to ANDROID LAYOUT FRAMEWORK by means of three scenarios.

The paper is structured as follows. We list and describe the layout frameworks in Section II followed by the related work in Section III. Next, we describe the experimental set-up of our case study in Section IV, which leads to results in Section V. Finally, we conclude the paper in Section VI.

II. Layout Frameworks

In this section, we list and describe the five layout frameworks under investigation in this study. Selected as we perceive these to be the five most popular frameworks.

A. Android Layout Framework

ANDROID LAYOUT FRAMEWORK is the underlying basic framework to develop Android applications. The framework provides a set of *view objects* to create the UI layout. A *view object* or a *widget* is a basic UI element. The UI layouts can be specified in both XML or defined in Java code. The layout defined via XML is inflated to an equivalent Java view hierarchy and provides a clear separation between layout definition and view implementation. Android Studio, the default IDE, provides a layout editor and layout preview allowing the developer to design layouts from within the IDE and simultaneously render the layout without having to deploy the application to a device—what you see is what you get (WYSIWYG). On the other hand, the layout written in Java code skips the I/O overhead of reading the XML file, parsing the XML, and the reflection involved when creating the *view objects* defined in the XML. Moreover this also eliminates the need for FindViewById calls, which are needed for *view object* lookups when the layout is specified in XML.

The layout specified in XML is first inflated and then created. During *layout inflation*, a definition of a layout hierarchy is converted to an in-memory object representation of that hierarchy. While in *layout creation*, the previously inflated layout is finally rendered on the screen. The underlying steps to ultimately render the UI layout on screen with Android Layout Framework include, *layout*, *measure*, and *draw*. In the *laying-out*, positional constraints on all *view objects* are resolved and used to compute their position on the screen. Then, in the *measure* step, the size constraints are resolved and used to compute the width and height of *view objects*. Finally, using these constraints, the *view objects* are rendered on the screen during the *draw* step. Android Layout Framework carries out all these steps sequentially in a single thread, often referred to as the main thread or UI thread.

B. Data Binding

Data Binding is a first party layout framework provided by Google [4]. It builds on top of the good parts of XML layouts such as the specialised tooling, layout preview and editor inside the IDE. It also provides a few solutions to its issues.

More specifically, Data Binding allows to insert Java code directly inside the XML. This can, for instance, be used to bind a Java attribute to the text attribute of a TextView object inside the XML. Then, whenever the Java attribute is updated, the change automatically propagates to the *view object*.

To boost performance, Data Binding only solves a single minor issue—FindViewById. The Data Binding library automatically converts all *view object* identifiers to *view object* references, which then can be used inside Java code. This conversion happens at compile time. Moreover, Data Binding also allows bindings for custom created *view objects* besides basic *view objects*.

C. Anko

Anko is a third party library provided by JetBrains[1], the creators of the Android Studio IDE [13]. Anko uses a domain-specific language (DSL) to create layouts in code written in Kotlin[2] and can only be used when creating an application in Kotlin—JetBrains' own JVM language that compiles down to Java bytecode.

Anko's DSL has many similarities with regular XML layouts. The layout definitions are concise and use code block nesting to mimic the hierarchy of the created layout. Anko can be seen as syntactic sugar over layout creation in Java code.

D. ConstraintLayout

ConstraintLayout is hardly a framework, yet it certainly is an improvement that enables developers to write extremely complex layouts using a single parent container. It was designed to replace nested RelativeLayouts and LinearLayouts [3]

ConstraintLayout, however, introduces new challenges related to flattening a layout to a single parent container. Replicating each and every behaviour that could previously be achieved by nesting several instances of LinearLayout and RelativeLayout can be difficult and very time consuming to replicate with ConstraintLayout. As it is only a new *view object*, it can actually be used in combination with any of the above frameworks.

E. Litho

Litho is a new layout framework, developed by Facebook, built with intention to completely replace the Android Layout Framework—a feat no other framework has attempted before [18]. While most of the previous solutions only go as far as modifying the way the *view objects* are created, Litho goes several steps further and defines its own set of *view objects*, which are mostly identical to the Android Layout Framework *view objects*. Moreover, Litho also takes care of the *measure*, *laying-out*, and *draw* by itself. Even when the Litho layout specifies hundreds of nested *view objects*, the resulting layout of Litho is drawn via a single Android Layout Framework *view object*.

Litho *view objects* only contain pure functions with immutable arguments making them thread-safe by default, which enables several asynchronous operations. While Android Layout Framework does every step of the layout creation (*laying-out*, *measure*, and *draw*) on the main thread, Litho can offload the *laying-out* and *measure* to a separate thread and free the main thread for handling other events. Thus, it is the only framework that does not fully rely on the Android layout renderer. It includes its own engine for the *laying-out* and *measure* steps, which are asynchronously performed in separate threads. This complete control over the layout creation enables Litho to heavily optimise the way the layout is created. For example, the RecyclerView recycles list items, Litho recycles *view objects* inside the list item layout on a per *view object* level; at even deeper level and for any layout not just the list items.

Like Anko, Litho also has its own DSL to define the layouts. However, unlike Anko that requires writing in Kotlin, Litho uses plain old Java code. There is, however, once again a learning curve as the constraints of Litho DSL have no resemblance with any of the existing Android constraints. Litho uses Facebook's Yoga library under the hood which is based on the CSS flexbox definition [20]. The learning curve of Litho is steep as its DSL has little resemblance with the Android Layout Framework.

III. Related Work

Plenty of research has been done on different app performance issues in Android apps [11], [15], [16], [24]. Most of the work includes developer surveys that aggregate the most popular issues, some of them also provide ways to combat these issues. Even books have been written on the subject [19]. Following is an overview of the most frequently discussed and related layout performance issues.

Blocking the UI Thread. Surveys by Linares-Vasquez et al. [15] and Liu and Cheung [16] count Blocking of the UI Thread as one of the main issues related to GUI lagging. Long running operations on the main thread cause the UI to freeze until the blocking operation completes. Nilsson has shown that delays shorter than 100 milliseconds feel instantaneous to

[1]https://www.jetbrains.com
[2]https://kotlinlang.org

the user, anything longer will break the illusion and will be noticeable to the user [17].

Complex Layout Hierarchies. Nilsson as well as Guy associate complex layout hierarchies to poor performance [11], [17]. Moreover, inflating and creating a layout with many different nodes with many constraints can take a lot of time. Slowdown during layout inflation and creation is caused by XML parsing, reflection , object allocation, *measure*, *laying-out*, and *draw* steps [21].

Double Taxation. Some container widgets in the Android framework require two layout passes even for their descendants in certain conditions—*double taxation*. A container suffering from *double taxation*, when nested, further increases the number of layout passes resulting in an exponential increase down the tree [1], [5]. Shirazi et al., in an empirical study, found LinearLayout and RelativeLayout as the most popular containers. Furthermore, they also found that LinearLayout appears frequently nested inside published apps while RelativeLayout is less frequently nested. Both of these suffer from *double taxation*. There is always a double layout pass for RelativeLayout, however, for LinearLayout it is only when *view objects* with weight attributes are used in the horizontal orientation.

List Scrolling. Liu and Cheung also include list scrolling performance as one of the issues related to GUI lagging [16]. The list may use internally either a ListView or a RecyclerView. Both *view objects* are designed to inflate a minimal amount of layouts to fill the visible section inside the device's viewport—an optimisation which is referred to as *view recycling*. ListView does not optimise view recycling by default, hence the ViewHolder design pattern needs to be used to fully reuse views. However, RecyclerView has solved this by integrating the ViewHolder design pattern and exposing only its necessary counterparts [2], [7].

IV. Experimental Set-up

Our case study explores performance improvements in single layout rendering. We measure the time needed to inflate a layout followed by the creation of this layout.

While we previously mentioned some of the most common layout performance pitfalls, it is important to clarify these do not apply to the listed experiments as we carefully crafted each layout making sure each layout was optimised as much as possible. It is however possible that some alternative frameworks handle these issues better in cases where it was impossible to completely work around a pitfall.

A. User Interface Layouts

To measure the actual performance improvements of a single layout we had to design a set of benchmarks that test the different performance issues that were mentioned in the Android Layout Framework.

To observe the actual performance improvements of each framework we designed two realistic UI layouts of subtly different complexities yet kept shallow in depth. The first one is a simple form with 5 input fields, 2 buttons, and some text

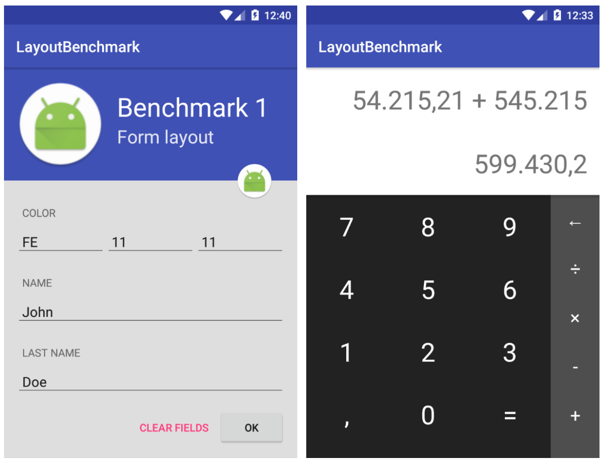

Fig. 1: Form layout (left) and Calculator layout (right).

labels. The second UI was inspired by the default calculator application, which is shipped with Android devices (Figure 1). Note that we write these layouts optimised in a reasonable manner using each framework to measure their corresponding timings.

B. Scenarios

Layout inflation. The time taken to inflate a layout from its definition is measured. The following two application states were benchmarked for *layout inflation*.

The cold start state. This is the state when the application is launched for the first time. This implies that the initialisation phase of the frameworks counts here. For repeated measurements, we kill the application and start a fresh instance.

The secondary cold start state. This is rather a more general use case. Here, the application is active and a new screen is launched causing a completely new layout to be inflated. This event happens all the time during the use of an application. Slowdowns during *layout inflation* are easily noticeable here. A faster *layout inflation* will also speed up switching between screens in the application translating into improved user experience. The idea here is to capture and explore the caching capabilities of each framework across independent layout specifications. To ensure an empty cache, we first kill the application. Next, once the first layout is inflated, we free that layout and inflate a completely different layout. We then only capture the duration of the second *layout inflation*.

Layout creation. Here, we measure the time it takes to create (render on screen) the inflated layouts. The *layout creation* consists of three steps: *measure*, *laying-out*, and *draw*.

C. Performance Measurement

Timing the *layout inflation* phase is trivially simple. As this operation is single threaded the thread is blocked until the layout is inflated. Thus, we easily calculate the duration of the code executed during inflation from start to finish using the wall clock time.

H/L \ $H_?$	Data Binding	Litho	Anko	Constraint Layout
Emu Form	H / H_1	H / H_0	L / H_0	H / H_1
S3 Form	H / H_1	L / H_0	H / H_0	H / H_1
Emu Calc	H / H_1	H / H_1	L / H_0	H / H_1
S3 Calc	H / H_1	H / H_1	H / H_1	H / H_1

TABLE I: Accepted Hypothesis using a maximum P-value of 0.01 and Higher/Lower average (lower is better) than ANDROID LAYOUT FRAMEWORK for *layout inflation*: the *cold start state* scenario.

On the other hand, to measure timings for *layout creation*, we use *Hierarchy Viewer*. It provides the times taken by the *measure*, *laying-out*, and *draw* steps of *layout creation* separately. Doing so independent from the *layout inflation* time. *Hierarchy Viewer* is a GUI application available starting from Android 4.1 with no command line interface [6]. The application is written in Java and exposes an API to collect the required information. Leveraging this API, we create our own command line interface to collect the timings. To create a robust measurement, we collect the timings of 50 executions for each event.

D. Devices

As argued by Vergauwen, performance improvements are more noticeable on slower devices [22], hence we use a Samsung Galaxy S3 mini running Android 5.1 as an older device. We use Genymotion, an emulator, also running Android 5.1 with 512 MB of RAM to mimic a high performance device.

V. RESULTS

We present the results in order, first for single layout followed by list layout. Each presented benchmark result for each alternative layout framework is verified for statistical significance compared to ANDROID LAYOUT FRAMEWORK using Welch's t-test [23]. We will utilise the following Hypotheses with a maximum P-value of 0.01:

H_0 – The two benchmark result populations have equal means.

H_1 – One of the benchmark result population means is greater than or equal to the other.

We will only be able to conclude a performance improvement if the null hypothesis is rejected while the benchmark result mean for the alternative framework is better than the mean of ANDROID LAYOUT FRAMEWORK.

A. Layout Inflation

Here, first we present the performance of frameworks for *layout inflation* measured for the different states of each application.

The cold start state. Figure 2 provides the *layout inflation* times for the *cold start state* scenario for two apps executed

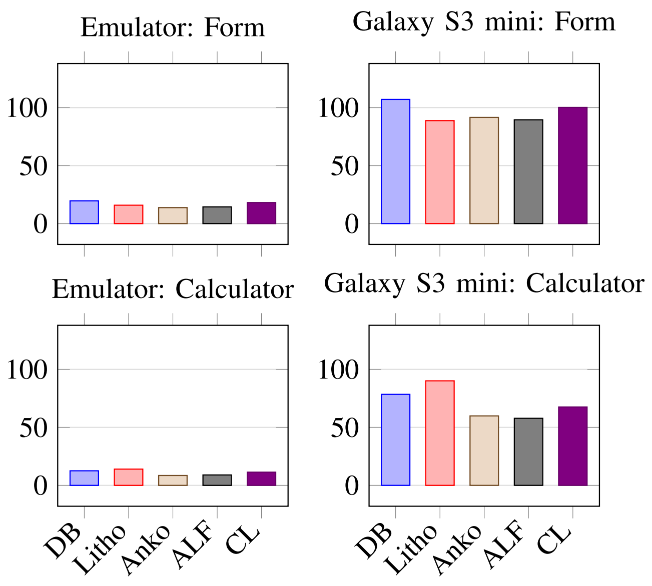

Fig. 2: Performance assessment for *layout inflation*: the *cold start state* scenario. Lower is better.

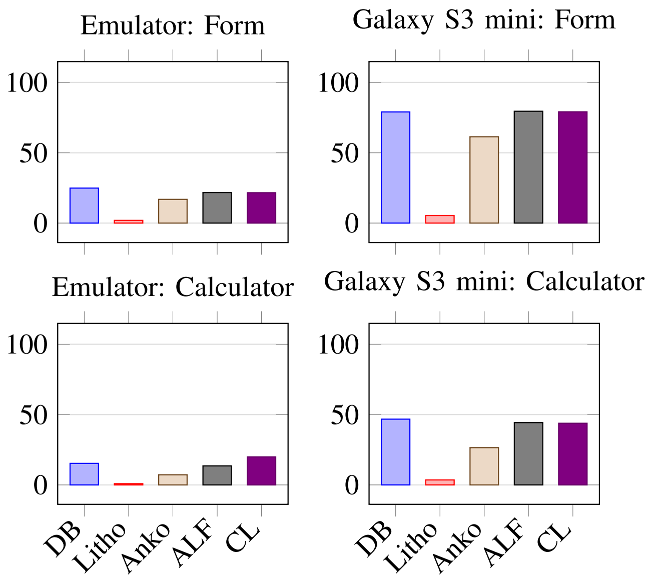

Fig. 3: Performance assessment for *layout inflation*: the *secondary cold start state* scenario. Lower is better.

on two devices. All the frameworks are faster with negligible performance difference for both apps on the emulator. However, the frameworks are comparatively slower on the slower S3 device for both apps. There, we observe that the ranking of ANDROID LAYOUT FRAMEWORK (ALF), DATA BINDING (DB), ANKO, and CONSTRAINTLAYOUT (CL) is fairly consistent for both apps, DATA BINDING being the slowest. LITHO, on the other hand, seems to behave inconsistently over the different apps. For the calculator app where other frameworks are faster, LITHO is slower which can be explained as the framework's initialisation

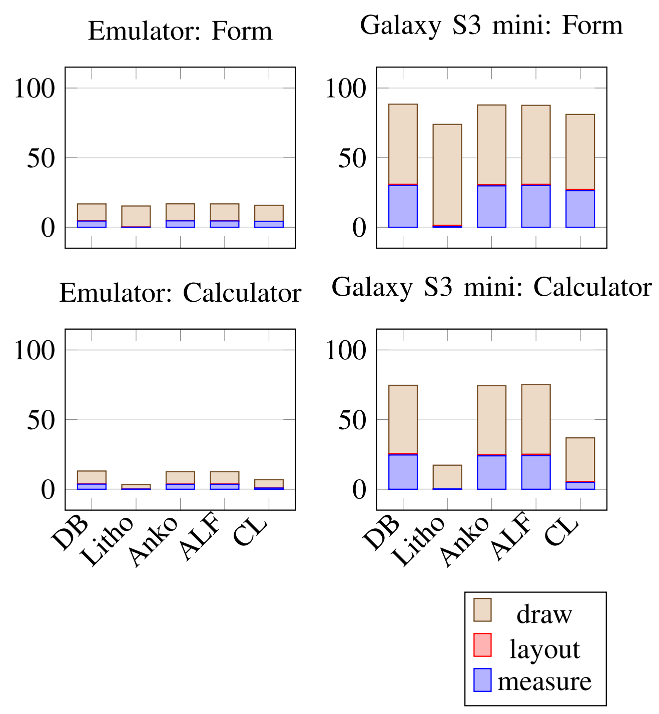

Fig. 4: Performance assessment for *layout creation*. Lower is better.

H/L \ $H_?$	Data Binding	Litho	Anko	Constraint Layout
Emu Form	H_0 / H	H_1 / L	H_1 / L	H_0 / L
Emu Calc	H_0 / H	H_1 / L	H_1 / L	H_0 / H
S3 Form	H_0 / L	H_1 / L	H_1 / L	H_0 / L
S3 Calc	H_1 / H	H_1 / L	H_1 / L	H_0 / L

TABLE II: Accepted Hypothesis using a maximum P-value of 0.01 and Higher/Lower (lower is better) average than Android Layout Framework for *layout inflation*: the *secondary cold start state* scenario.

H/L \ $H_?$	Data Binding	Litho	Anko	Constraint Layout
Emu Form	H_0 / L	H_1 / L	H_0 / L	H_1 / L
Emu Calc	H_0 / L	H_1 / L	H_0 / L	H_1 / L
S3 Form	H_0 / L	H_1 / L	H_0 / H	H_0 / L
S3 Calc	H_0 / H	H_1 / L	H_0 / L	H_1 / L

TABLE III: Accepted Hypothesis using a maximum P-value of 0.01 and Higher/Lower average (lower is better) than Android Layout Framework for *layout creation*.

overhead.

When we consider the results of the statistical test in Table I it however becomes clear that the observed performance improvements are either too insignificant to reject the null hypothesis or produce an average that is worse than Android Layout Framework. Out of all the results there isn't a single result that both rejects the null hypothesis and has a lower average. There are however a high number of results where the null hypothesis is rejected with a higher average, indicating a negative performance impact. Data Binding, Litho and ConstraintLayout are all affected by this. We can therefore conclude that performance improvements for the *cold start state* scenario are non-existing for Anko and even adverse for the other three alternatives.

The secondary cold start state. Figure 3 shows the performance of the frameworks exposing the caching capabilities of each framework across independent layout specifications. Here, we see that all frameworks reveal some performance improvement on the slower S3 device. Amongst all, Litho takes close to 0 milliseconds to inflate the layout indicating it has comparatively the best caching mechanism, while Anko comes second and the other three have almost the same performance. The performance improvement of Litho is more prominent on the slower device, where it takes close to 0 milliseconds when it previously had a constant time for the *cold start state* scenario.

When we verify these observations using the statistical test results presented in Table II we see that indeed Anko and Litho both reject the null hypothesis in all tests while obtaining an average lower than Android Layout Framework. We can conclude that for the *secondary cold start state* scenario both Anko and Litho present real world performance improvements over Android Layout Framework. For the other alternative frameworks there isn't a single entry where both the null hypothesis is rejected and a lower average is achieved. For these two frameworks performance improvements remain non-existent or even adverse.

B. *Layout Creation*

Now we present the performance of each framework to measure the time taken to create the previously inflated layouts. Note that *layout creation* consists of three steps: *measure*, *laying-out*, and *draw*.

The time duration of each step is given in Figure 4 for each device and app. Here the best performer is Litho closely followed by ConstraintLayout (CL). The performance improvements of ConstraintLayout and Litho are most noticeable in the Calculator app. As the *measure* and *layout* are performed in separate threads by Litho, these threads effectively halt the *draw* operation until they complete and join the main thread. Thus, the actual *measure* and *layout* times for Litho are included in the *draw* time, resulting in a slightly longer *draw* time yet an overall shorter *render time*.

On the other hand Android Layout Framework (ALF), Data Binding (DB), and Anko have the same performance. This is because they each use the same underlying procedures of ANDROID LAYOUT FRAMEWORK for *layout creation*. Moreover, each of them constructs an identical layout tree, hence send an identical collection of *view objects* to the renderer.

We verify these findings using the statistical test results in Table III where we see both a rejected null hypothesis and a lower average for the calculator layout for both LITHO and CONSTRAINTLAYOUT while none of the other frameworks reject the null hypothesis. Similar results were obtained for the form layout. For DATA BINDING and ANKO we obtained high P-values resulting in accepted null hypotheses indicating insignificant differences in performance as expected.

VI. CONCLUSION

We compared the performance of four alternative Android layout frameworks (namely, DATA BINDING, ANKO, CONSTRAINTLAYOUT, and LITHO) to ANDROID LAYOUT FRAMEWORK by means of three different scenarios. We experimentally investigated the impact of *layout inflation* (with starting states cold vs. warm) and *layout creation* for single layout, both critical steps in rendering the GUI of an app. We verified these findings by applying a statistical t-test.

When measuring the performance during *layout inflation*, we observed a negative performance impact for three out of the four alternative frameworks in the *cold start state*. In a warm state, both LITHO and ANKO perform notably better even on slower devices due to their caching capabilities. For the *layout creation* scenario we noticed positive performance improvements for both LITHO and CONSTRAINTLAYOUT.

We conclude that the frameworks vary a lot in their performance over the presented scenarios. Nevertheless, there is no single best solution suggesting that hybrid combinations are worthwhile to investigate in the future.

REFERENCES

[1] Britt Barak. 2016. Layout Once, Layout Twice — Sold! (2016). https://medium.com/@britt.barak/layout-once-layout-twice-sold-aef156ff16a4

[2] Android Developers. 2016. Making ListView Scrolling Smooth. (2016). https://developer.android.com/training/improving-layouts/smooth-scrolling.html

[3] Android Developers. 2017a. Build a Responsive UI with ConstraintLayout. (2017). https://developer.android.com/training/constraint-layout/index.html

[4] Android Developers. 2017b. Data Binding Library. (2017). https://developer.android.com/topic/libraries/data-binding/index.html

[5] Android Developers. 2017c. Performance and View Hierarchies. (2017). https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html

[6] Android Developers. 2017d. Profile Your Layout with Hierarchy Viewer. (2017). https://developer.android.com/studio/profile/hierarchy-viewer.html

[7] Android Developers. 2017e. RecyclerView. (2017). https://developer.android.com/guide/topics/ui/layout/recyclerview.html

[8] Android Developers. 2018. Improving Layout Performance. (2018). https://developer.android.com/training/improving-layouts/index.html

[9] Andrew Drobyazko. 2016. Performance comparison - Building Android UI with code (Anko) VS XML Layout. (2016). https://nethergrim.github.io/performance/2016/04/16/anko.html

[10] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Mining Test Repositories for Automatic Detection of UI Performance Regressions in Android Apps. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 13–24. DOI:http://dx.doi.org/10.1145/2901739.2901747

[11] Romain Guy. 2009. Turbo-charge your UI. *Google Inc., Google I/O* (2009).

[12] Statista Inc. 2017. Number of apps available in leading app stores as of March 2017. (2017). https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores

[13] Jetbrains. 2017. Anko. (2017). https://github.com/Kotlin/anko

[14] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. 2015. What Do Mobile App Users Complain About? *IEEE Software* 32, 3 (May 2015), 70–77. DOI:http://dx.doi.org/10.1109/MS.2014.50

[15] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How Developers Detect and Fix Performance Bottlenecks in Android Apps. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 352–361. DOI:http://dx.doi.org/10.1109/ICSM.2015.7332486

[16] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1013–1024. DOI:http://dx.doi.org/10.1145/2568225.2568229

[17] Elin Nilsson. 2016. *A Recipe for Responsiveness : Strategies for Improving Performance in Android Applications*. Master's thesis. UmeåUniversity, Department of Applied Physics and Electronics.

[18] Lucas Rocha. 2016. Components for Android: A declarative framework for efficient UIs. (2016). https://code.facebook.com/posts/531104390396423/components-for-android-a-declarative-framework-for-efficient-uis/

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

[20] Emil Sjölander. 2016. Yoga: A cross-platform layout engine. (2016). https://code.facebook.com/posts/1751945575131606/yoga-a-cross-platform-layout-engine/

[21] Cătălin Tudose, Carmen Odubăşteanu, and Serban Radu. 2013. *Java Reflection Performance Analysis Using Different Java Development*. Springer Berlin Heidelberg, Berlin, Heidelberg, 439–452. DOI:http://dx.doi.org/10.1007/978-3-642-32548-9_31

[22] Simon Vergauwen. 2016. 400% faster layouts with Anko. (2016). https://android.jlelse.eu/400-faster-layouts-with-anko-da17f32c45dd

[23] B. L. WELCH. 1947. THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. *Biometrika* 34, 1-2 (1947), 28–35. DOI:http://dx.doi.org/10.1093/biomet/34.1-2.28

[24] S. Yang, D. Yan, and A. Rountev. 2013. Testing for poor responsiveness in android applications. In *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*. 1–6. DOI:http://dx.doi.org/10.1109/MOBS.2013.6614215