FUEL: A Runtime Methodology to Preload Time Consuming UI-APIs for Android Apps

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Zheng Cui, B.S

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2020

Thesis Committee

Xiaorui Wang, Advisor

Irem Eryilmaz

Abstract

It is important for the developers to keep their applications responsive. However, a complex user interface (UI) can cause responsiveness problems during the execution of UI-APIs [1]. Even worse, a soft hang may be generated when the execution time of a UI-API is longer than a perceivable delay (i.e., 100ms). This paper presents FUEL (Fast UI Elements Loading), a runtime methodology that preloads the UI elements that are likely to be executed in order to reduce the response time. FUEL consist of three components: the view cache, the preloading module and the prediction module. It records the loading sequence of activities with complex UI at runtime and predicts which UI layout should be inflated ahead of time. After inflation, the generated views are saved in the view cache and can be retrieved through its corresponding class name when needed. We have tested FUEL on four open source applications that are available in the Google Play Store or on GitHub. As the results show, FUEL is able to reduce the execution time of UI-APIs by at least 28%, thus to eliminate soft hangs caused by complex UIs.

Vita

2018.....B.S. Electrical Engineering, University of Electronic Science and Technology of China

2018 to present.....Department of Electrical and Computer Engineering, The Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

An important feature of smartphone applications is the complex user interfaces (UI). However, a complex user interface can cause responsive problems during the execution of UI-APIs (Application Programming Interface). Even worse, a soft hang may be generated when the execution time of a UI-API is longer than a perceivable delay (i.e., 100ms [2]). The user experience may be extremely bad if the application has too many responsive problems. The motivation of this project is to reduce the execution time of UI-APIs without modifying the UI elements. This paper presents FUEL (Fast UI Elements Loading), a runtime methodology that executes heavy UI operations ahead of time and caches preloaded UI elements in the memory. All the design and test are implemented on Android platform. The rest of this paper is organized as follows. Chapter 2 introduces background about Android development, especially about UI. Chapter 3 describes the design and implementation of FUEL. Chapter 4 shows the performance of FUEL on open source applications. Chapter 5 concludes the paper.

Chapter 2. Background

2.1 Android Platform Architecture

Android is an open-source operating system based on the Linux kernel. Figure 1 is from the official documentation of Android and it shows the main components of the Android platform [3].

2.1.1 The Linux Kernel

The Linux kernel is the foundation of Android platform. It provides underlying drivers for various hardware of devices, such as camera drivers, Wi-Fi drivers etc., and power management. Also, Android Runtime (ART) relies on the Linux kernel to perform functions such as threading and low-level memory management. The security mechanism of the Linux kernel provides corresponding protection for Android.

2.1.2 Hardware Abstraction Layer (HAL)

The hardware abstraction layer (HAL) provides interfaces to Java API Framework. The HAL contains multiple library modules. When an API requests access to a specific hardware of device, the corresponding library module is loaded by Android system.

2.1.3 Android Runtime (ART)

For devices running Android 5.0 (API level 21) or higher, Android Runtime (ART) enables each application to run in its own process. ART is specially customized for mobile devices, and it is optimized for limited memory and CPU of mobile phone.

2.1.4 Native C/C++ Libraries

Native C/C++ Libraries are required by Android system components and services because they are built from native code written in C and C ++. Developers can use Java

framework APIs to access these native libraries and add different support in their applications. For example, the SQLite library provides database support, the OpenGL ES library provides 3D drawing support, and the Webkit library provides browser kernel support.

2.1.5 Java API Framework

It provides various APIs (Application Programming Interface) that may be used when building applications. Some Android system applications are developed using these APIs. Developers can also build their own applications with these APIs.

2.1.6 System Applications

It includes all the applications installed on mobile phone, such as contacts, calendars, internet browsing, and other applications belonging to Android system, and of course, all third-part applications.

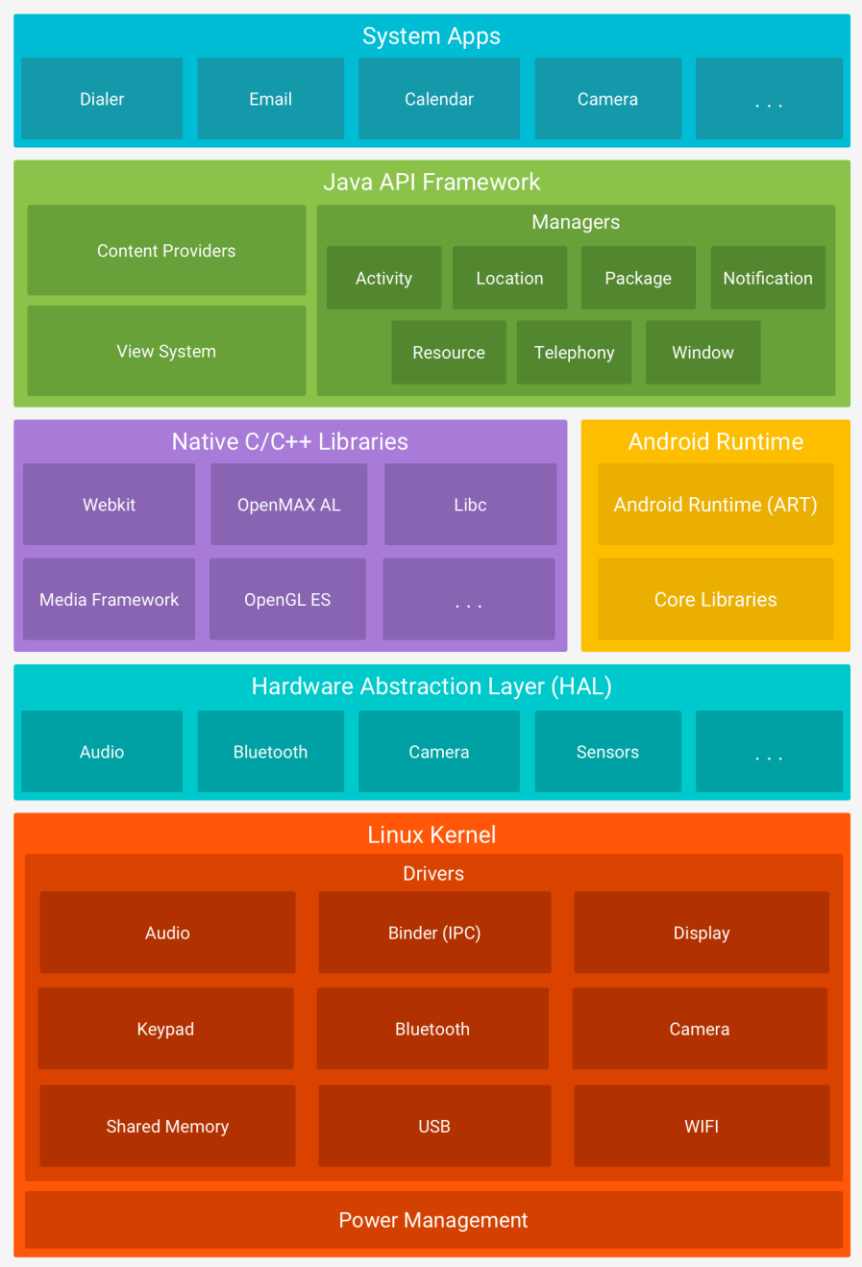


Figure 1. Components of the Android platform [3].

2.2 Introduction to Android Development

2.2.1 Major Components of an Android Application

The four major components of the Android system are Activity, Broadcast Receiver, Service, and Content Provider, which are frequently used to build an application. But not every Android application consists of all four components. Developers can use the entire feature set of the Android OS (Operating System) through APIs written in the Java language to build them. The activity class is the most important building block in Android development. Almost all Android applications interact with users through activities, for this reason the activity class is mainly responsible for creating display windows that contain information, buttons, pictures, etc. It is noteworthy that the development of Android applications pays attention to the separation of logic function design and UI design. It is not recommended to write the user interface directly in an activity. A more general approach is to define the UI in a layout file and then place it in the corresponding activity with *setContentView()* method. A service is an application component that can run in the background performing some long-time operations, and it does not directly interact with the user. For example, a service can perform tasks like playing music, downloading, file I/O and so on. A service can be started by other application components. It will run in the background until its operations end or it is stopped by the application or system. Note that a service can continue to run even if the user switches to another application or exits the application. A broadcast receiver is used to receive broadcast messages from various places, such as the Android system and other applications. Broadcast messages can be sent with *Context.sendBroadcast(Intent)* method

if an event of interest occurs. When a Broadcast Receiver is triggered, the system will perform corresponding operations, for example launching a needed application. The content provider class makes it possible to share data between applications. Content providers can also help an application to access the data saved in files, SQL (Structured Query Language) databases, and other storages. For example, the information of contacts can be read through a content provider. All components of an application, which include all activities, services, broadcast receivers, and content providers, must be declared in *AndroidManifest.xml* file after implementation. The manifest file contains essential information about an application (e.g., the application's package name, the components of the application, the required user permissions).

2.2.2 Activity

As almost all Android applications interact with users through activities and the UI of display windows is mainly loaded in activities, more information about the activity class is introduced in this section.

When an application is launched, its main activity will be started and display the interface on the screen. Main activity is declared in *AndroidManifest.xml* file, and almost every android application contains one main activity. Every time a new activity is started, it will be placed over the previous activity. When it is destroyed, the previous activity will return to foreground. Actually, all activities are managed in activity stacks by the Activity Manager of Android OS. The stack is a last-in-first-out data structure. By default, whenever the user starts a new activity, it will be pushed into the stack and placed at the top. If the user presses the back button or call the *finish()* method to destroy the top

activity, it will be popped out of the stack, and the activity below it (i.e. the previous activity) will be at the top of the stack again. Activities in the stack are never rearranged. Figure 2 shows an example of how an activity stack works.

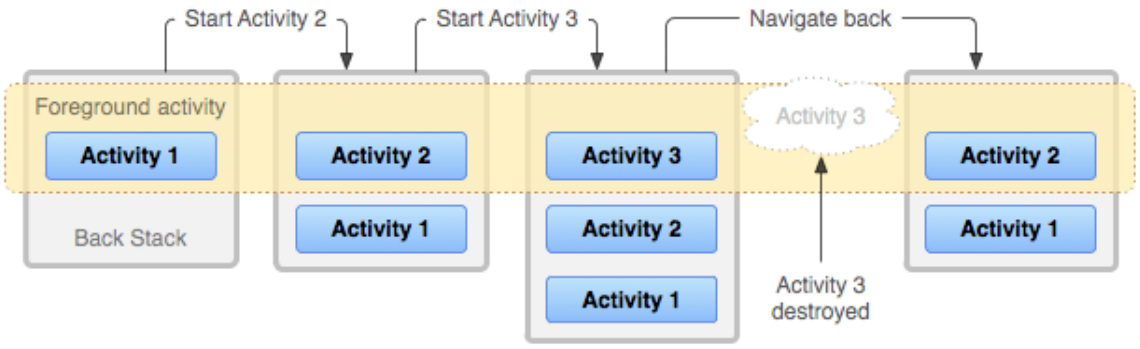


Figure 2. An example of activity stacks [4].

An activity has four states in its life cycle:

- **Running.** If an activity is at the top of activity stacks, it is active or running. This is usually the activity interacting with users.
- **Visible.** If an activity is no longer at the top of activity stacks but is still visible, the activity enters a visible state. It is possible if the running activity cannot fill the entire screen.
- **Stopped.** If an activity is no longer at the top of activity stacks and is completely invisible, it is stopped. Its data and variables will be saved by the system. But

activities in the stopped state may be killed and recycled by the system, when memory is needed elsewhere.

- **Destroyed.** If an activity is removed from activity stacks, it is destroyed. The system tends to recycle destroyed activities to ensure that the phone has sufficient memory.

Figure 3 is from the official documentation of Android. It shows the life cycle and state paths of an activity. There are seven callback methods (in rectangles) that can be used by developers to perform operations:

- *onCreate()*: It is called when the activity is first created. Some initialization operations should be completed in this method, such as loading UI, binding data to lists, etc.
- *onStart()*: It is called when the activity turns from invisible to visible.
- *onResume()*: It is called when the activity is ready to interact with the user. The activity at this time must be at the top of activity stacks and running.
- *onPause()*: It is called when the system is ready to start or resume another activity.
- *onStop()*: It is called when the activity is completely invisible to the user.
- *onDestroy()*: It is called before the activity is destroyed.
- *onRestart()*: It is called before the activity moves from stopped state to running state.

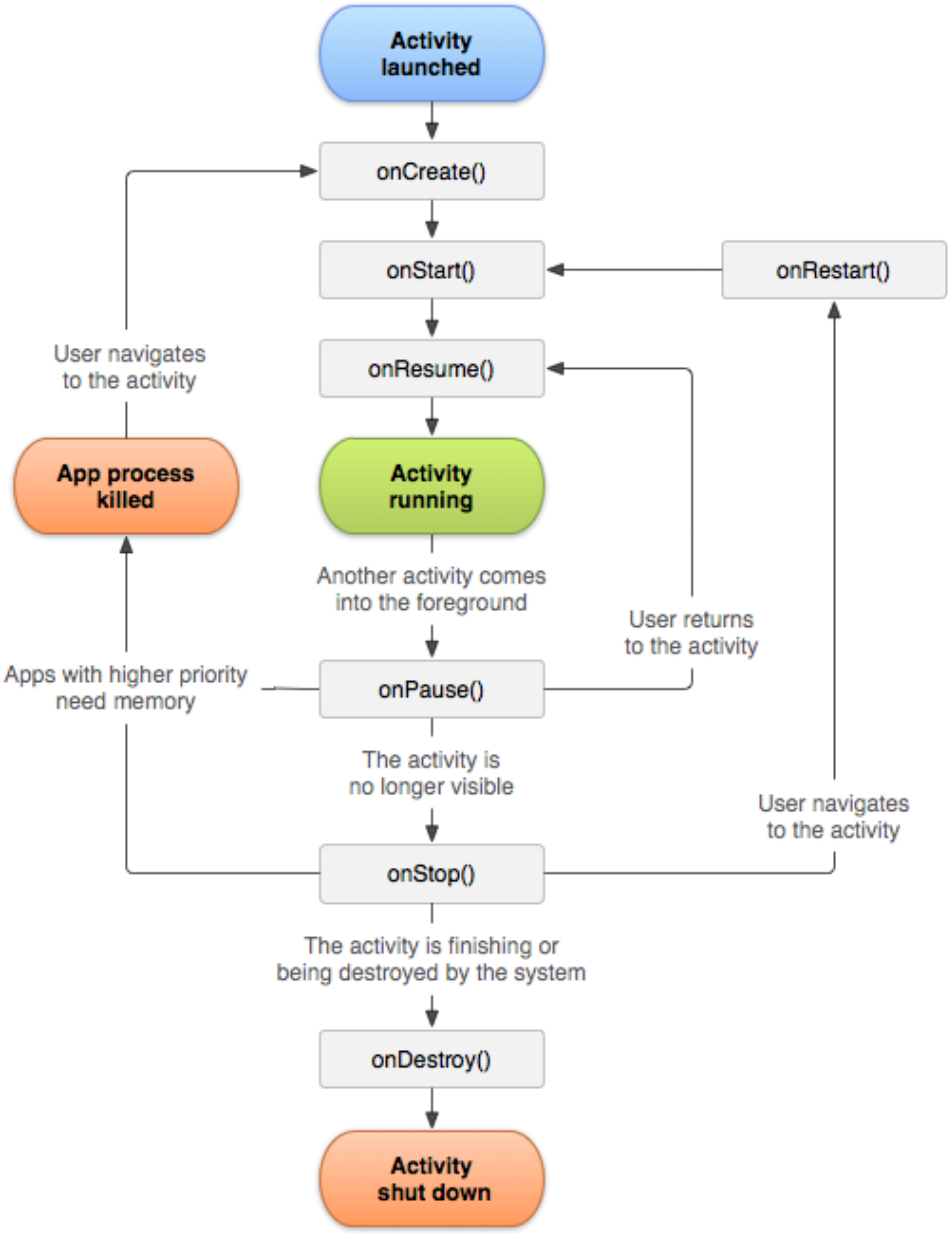


Figure 3. Activity lifecycle [5].

2.2.3 User Interface (UI)

A layout defines the structure of a user interface. In Android applications, all UI layouts are composed of View and ViewGroup objects. Figure 4 shows a hierarchy of UI layout. A View is an object that can be seen or interacted with the user, and it is usually called "widget". Android platform provides a large number of UI widgets. For example, a TextView displays text in the layout, a Button responds to click or tap event to perform an action, an ImageView displays an image to the user. Developers can also customize view objects and implement a custom view by overriding some of the standard methods called by the framework on the view. A ViewGroup is a layout container for other View and ViewGroup objects. Developers can use different layout structure types to build UI (e.g., LinearLayout, RelativeLayout, FrameLayout, etc.). A UI layout can be designed in two ways:

- **Declare UI elements in XML.** Android provides an XML vocabulary that corresponds to the View classes and subclasses. As mentioned above, it is recommended to separate logic function design and UI design. Developers can declare an entire UI layout in an XML file instead of in the activity class. Then a resource ID will be assigned to the layout file by the Resource Manager. It is easy to load the UI in the corresponding activity with *setContentView(int id)* method when the activity is created (i.e., when *onCreate()* method is called).
- **Instantiate layout elements at runtime.** View and ViewGroup objects can also be declared programmatically. This approach is usually for modifying UI elements at runtime.

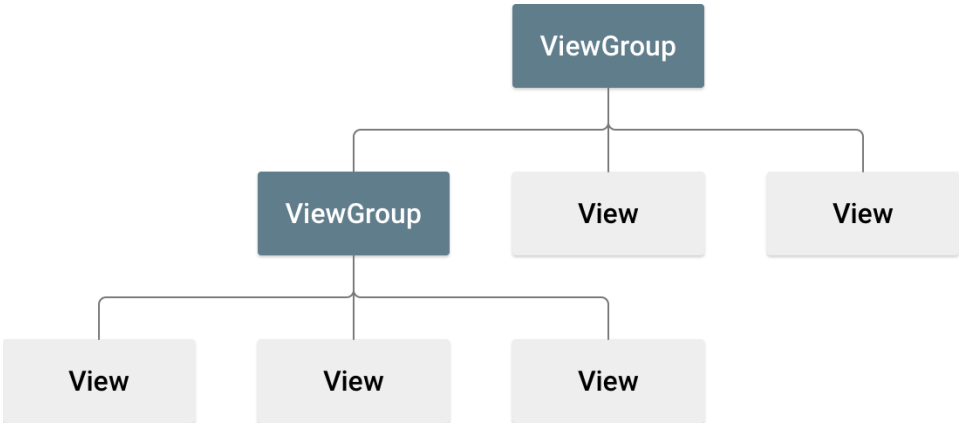


Figure 4. A hierarchy of UI layout [6].

2.2.4 Practical Example of UI Design

Figure 5 shows an entire UI layout of the *CropImageActivity* of the application *PDF Converter*. The user interface is designed for editing photos selected by the user. The XML file are shown in Figure 5(a), but some attributes of its layouts and widgets are hidden. Figure 5(b) shows the preview of the UI (left part) and the corresponding structure (right part). The UI layout is composed of four layouts (i.e., ViewGroups) and several widgets (i.e., Views). The outermost layout (in the white rectangle in Figure 5(b)) is CoordinatorLayout, which contains two other layouts: an AppBarLayout (in the green rectangle) and a LinearLayout (in the red rectangle). Only one Toolbar widget is in AppBarLayout. The LinnearLayout in the red rectangle contains a CropImageView widget customized by the developer, whose function is showing multiple selected

images, and another LinearLayout (in the yellow rectangle), which consists of two Button widgets, two ImageView widgets and one TextView widgets. The positioning of layouts and widgets in the XML file must be defined by their attributes and parameters. For example, there are four attributes in the innermost LinearLayout (in the yellow rectangle):

- *Layout_width*: This attribute defines the width of the LinearLayout. Its value is *match_parent*, which means the width matches that of its parent layout (i.e., the width of screen in this example).
- *Layout_height*: This attribute defines the height of the LinearLayout. Its value is *warp_content*, which means the height of the layout can just contain the inside elements.
- *Layout_marginTop*: This attribute defines the width of top margin, 5dp in this example.
- *Orientation*: This attribute defines the orientation of the inside elements. In this example, *"horizontal"* means the inside widgets are arranged horizontally. The positioning of the inside widgets (i.e., buttons, textView and imageViews) is defined by the sequence and attributes of them.

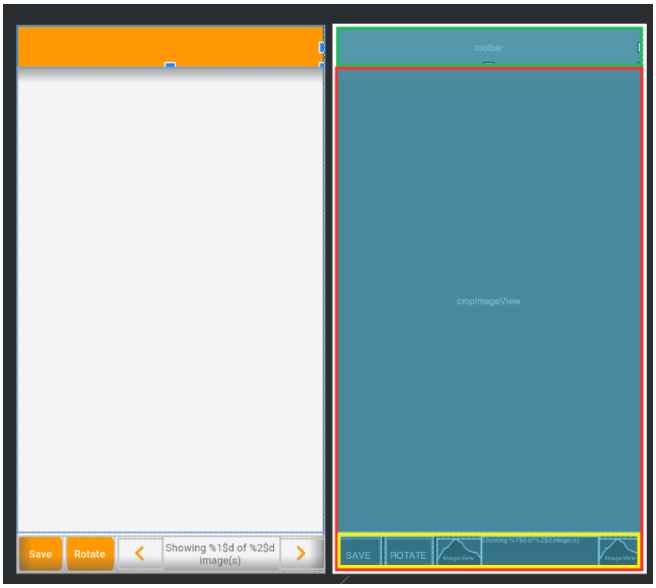
This UI layout is loaded in the *onCreate*() method of the CropImageActivity with *setContentView(R.layout.activity_crop_image_activity)*. When the activity is created, its *onCreate*() method will be called and the UI will be loaded at this time.

```
<android.support.design.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:custom="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".activity.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <android.support.v7.widget.Toolbar...>

    </android.support.design.widget.AppBarLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="?android:attr/actionBarSize"
        android:orientation="vertical"
        tools:context=".activity.CropImageActivity">

        <com.theartofdev.edmodo.cropper.CropImageView...>

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="5dp"
            android:orientation="horizontal">

            <Button...>

            <Button...>

            <ImageView...>

            <TextView...>

            <ImageView...>

        </LinearLayout>

    </LinearLayout>

</android.support.design.widget.CoordinatorLayout>
```

(a) continued

Figure 5. A UI layout of *PDF Converter*: Part (a) the layout XML file. Part (b) the interface shown on the mobile phone (left part) and the UI structure (right part).

Figure 5 continued



(b)

2.2.5 Information about UI-APIs

- *android.app.Activity.setContentView (View view)*: It loads and sets the UI for an activity. Its input parameter refers to an explicit view which is added to the activity's view hierarchy.
- *android.app.Activity.setContentView (int layoutResID)*: It loads and sets the UI for an activity. Its input parameter refers to the resource ID of a layout XML file. When calling this method, the layout XML file is converted to a view. Then this view is placed to the activity’s content.
- *android.app.Activity.findViewById(int id)*: It Finds a view by its ID defined by *android:id* attribute in XML file.

- *android.support.v7.app.AppCompatActivity.setContentView*: It loads and sets the UI for an activity, and has more operations compared to the *setContentView* method in Activity class.
- *android.app.Activity.addContentView*: It add an additional view to the activity's content without removing the existing content view.
- *android.app.Activity.startActivity*: It launches a new activity to show in the foreground.
- *android.app.Activity.finish*: It is called when an activity should be closed. It allows the activity to terminate any running job before calling *onDestroy*().
- *android.view.LayoutInflater.inflate (int resource, ViewGroup root)*: It coverts an layout XML file into a corresponding view hierarchy. The first parameter *resource* refers to the resource ID of an XML file. The second parameter *root* is an optional ViewGroup to be the parent of the generated hierarchy. This value can be *null*. This method returns the created view object. Note, this method cannot be used directly. Instead, a LayoutInflater instance must be retrieved first.

2.3 The Reasons Why UI-APIs can Generate Responsiveness Problems

There are several possible causes of slow UI responsiveness:

- **Structure**: Each widget and layout added to the application requires initialization, layout inflation, and drawing. For example, using nested layout structure instead of LinearLayout can lead to an excessively deep view hierarchy. Furthermore, nesting several instances of LinearLayout can be especially expensive as each child needs to be measured more than once. This is particularly important when the layout is

inflated repeatedly, such as when used in a ListView or GridView. The more views the application has, the more time it will take to measure, layout, and draw. To minimize the time it takes, it is important to keep the UI component tree as flat as possible, and remove all views that is not essential.

- **Cold vs Warm start**: With cold start, the application is not in memory, and it needs to load the entire UI from scratch. However, if all the application's activities are resident in memory, the application can avoid to repeating object initialization, layout inflation, and rendering. This is the main target of application preloading [7].

Chapter 3. Methodology and Design

3.1 Motivations and Goals

As mentioned above, in order to make UI-APIs responsive, it is important to keep the UI's hierarchy flat (i.e., reduce the nesting structure and remove redundant views). However, application developers may inevitably design a complex UI to make the interface beautiful and attractive. The main target of FUEL (Fast UI Elements Loading) is to build a cache of heavy UI operations executed between consecutive user actions, so that the execution time of UI-APIs can be shorter when the user interacts with the application. FUEL runs at runtime on the users' devices and has two main goals:

- Preload UI elements asynchronously in the background and save them in cache.
- Decide which UI elements should be loaded ahead of time. Because preloading those UI elements that may not be interact with users leads to energy and memory waste [8].

3.2 Pre-execution of UI-APIs

Every time the UI of an application is updated, the application and the OS execute a series of operations to update the displayed views and/or generate an animation. Upon the execution of a UI-API the application executes the following jobs on the main thread:

1. Call the *setContentView(view)* method of the PhoneWindow class, the base class for a top-level window look and behavior policy. An instance of this class is the top-level view which provides some UI configurations such as title, background, etc.
2. Initialize *mContentParent*. This variable refers to the DecorView, the top-level view of the window (i.e. DecoreView in Figure 6), in which the UI of activity's content are placed. The attributes of DecorView are initialized in this step such as theme, visibility of the window, title, etc.
3. Layout inflation: convert the layout XML file into a corresponding view and add the inflated view to the DecorView with *inflate(layoutResID, mContentParent)* method.
    a. Retrieve an XmlResourceParser object through *getLayout(Resource)* method. The XmlResourceParser class defines a series of APIs for parsing XML files.
    b. Parse the layout XML file and inflate the root view.
    c. Inflate child views and attach them to the root view.
    d. Return the root view of the inflated hierarchy and add it into the DecorView.
4. Call *onContentChanged()* method of the activity to place the UI to activity's content, i.e., ContentView in Figure 6.

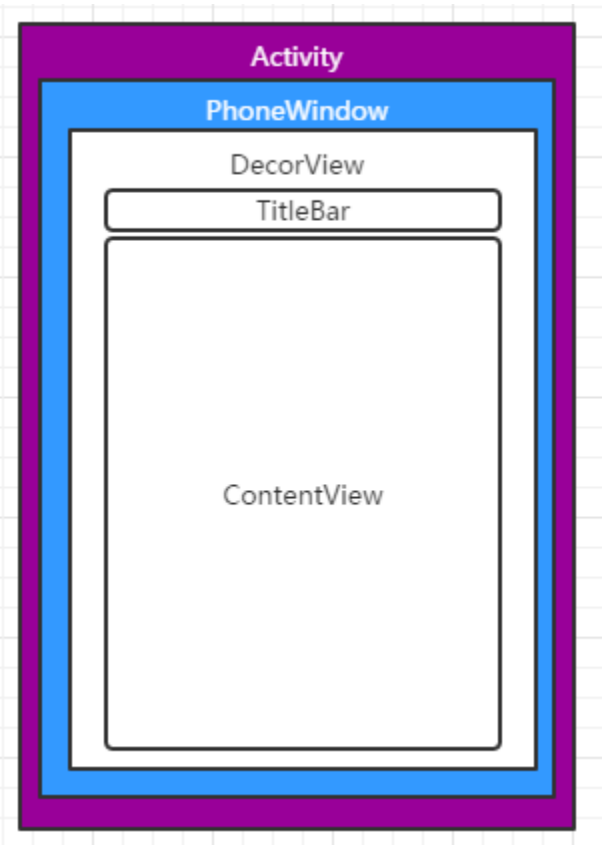


Figure 6. Top-level hierarchy of an activity's UI [9].

The UI-API *inflate()* is the UI-API that can cause responsiveness problems because it is computation intensive. The basic idea of FUEL is that executing the *inflate* APIs, i.e. above step 3, ahead of time so as to potentially reduce the response time of user actions. After inflation, the UI XML file of an activity will be converted into a view and this view will be saved into cache. When the activity is launched, we just need to place the preloaded view to activity's content without converting. Figure 7 shows the execution time of a UI-API of the application *InstaMaterial* with and without pre-execution. In the

original application, all UI operations of the activity, *UserProfileActivity*, are executed on the main thread when the activity is created with a response time of 62ms (labeled all UI operations in Figure 7). The execution time can be reduced to 38ms by preloading UI on another thread (labeled Set inflated views to activity's content in Figure 7). Specifically, the UI inflation, i.e. converting the layout XML file into views, is executed before creating the activity and the inflated views are saved in cache, which takes 21ms in total. When the activity is created, the remaining operations on the main thread are just retrieving the inflated views and setting them to activity's content.

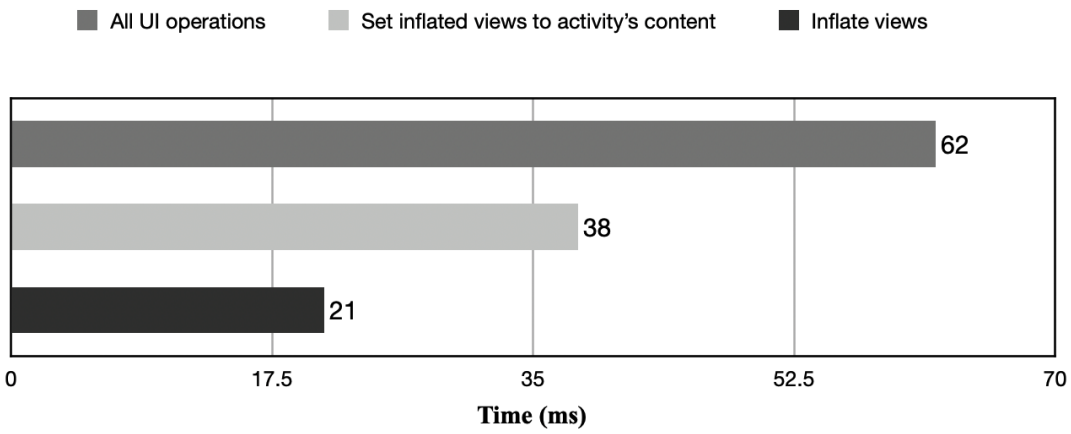


Figure 7. The execution time of UI-APIs of *InstaMaterial*.

3.3 Design of FUEL

3.3.1 Design Overview

FUEL has three main components: the view cache, the preloading module and the prediction module. FUEL works at runtime to decide which UI elements should be loaded ahead of time to decrease their response time. It uses a prediction algorithm that examined the application's current foreground activities and identifies which UI elements are likely to be executed before the application's contexts changes. The heavy UI operations of the identified UI elements are preloaded between user's actions on a new thread.

3.3.2 Preloading Module

The preloading module starts a new thread and inflate views from the XML layout file a head of time on that thread. It should be initialized when the application is launched. There is some required information stored in the module:

- **The resource ID of layout XML files.** It is the essential input variables needed by UI-APIs and should be entered in the initialization step. Note, there may be more than one resource ID, but each resource ID should not be identical to the others.
- **The class name of corresponding activities.** It is combined with the resource ID in the *PreloadInfo* class and should be entered with resource ID. Similarly, all names must be different from each other.
- **LayoutInflater.** Only one instance of LayoutInflater is saved in the preloading module. Used to inflate views, i.e. convert the XML layout file to view objects. It is retrieved from the main activity's context in the *onCreate()* state.

After initialization, the preloading module first creates a new thread by declaring a *Thread* class that implements the *Runnable* interface. Then override the *run()* method and implement all operations here. An Android application can have multiple concurrent threads executing different operations asynchronously. Thus, all the following operations run on this new thread in order to preload UIs and keep the main thread responsive. Once there are no preloaded UIs in the view cache, the prediction module decides which UI layout should be loaded ahead of time. After that, the LayoutInflater inflates views from the layout file by calling *inflate()* method (i.e. do Step 3 mentioned in section 3.2). Then the inflated views are saved in the view cache by calling *put(String name, View view)* method. The second variable refers to the inflated view, and the first variable refers to the class name of its corresponding activity. The inflated view is saved into cache with the class name mapped to it. More information is provided in the next section.

Usually, the view cache only keeps one inflated UI layout and the preloading module starts inflating a new layout if the old one is loaded by corresponding activity. Note, the prediction module can be invalidated, i.e. the preloading module runs without prediction. In this running mode, all UI layouts are inflated and kept by the view cache. It is recommended to use this mode if only one UI layout need to be preloaded in the application.

There are several methods declared in the preloading module:

- *start()*. Start the preloading module without prediction.
- *startInSequence()*. Start the preloading module with prediction.

- *setPreloadInfo(String name, int id)*. Save the essential information of a UI layout that need preloading. The first input parameter *name* refers to the class name of an activity. The second input parameter *id* refers to the resource ID of the activity's layout XML file.
- *setInflater(Context context)*. Retrieve LayoutInflater from the *context* of the application. The input parameter *context* can be an instance of the activity class or the application class.

3.3.3 View Cache

The view cache is a storage in which the inflated views are saved in. The main component is HashMap, a data structure that can map keys to values. The main advantage of using HashMap is the high performance of the operations such as adding, deleting, searching data in HashMap. The inflated views are saved into the HashMap with the class name of corresponding activity. There are several methods declared in the view cache:

- *put(String name, View view)*. The second variable refers to the inflated view, and the first variable refers to the class name of its corresponding activity (key). This method saves the input view into the HashMap and maps the name to the view. If a view in the cache is needed, it can be retrieved through its key (i.e. class name).
- *get(String name)*. Retrieve the inflated views in the cache. The input variable *name* refers to the class name of an activity (key). It returns the preloaded views to which the key is mapped.

3.3.4 Prediction Module

The prediction module examines the application's contexts (e.g., current foreground activity) and identifies which UI elements are likely to be interacted with the user. The basic idea of prediction is to record the loading sequence of interested activities (i.e., the activities whose UI elements need to be preloaded). The prediction module first registers an *ActivityLifecycleCallbacks* to monitor the lifecycle of all activities. Once an interested activity is launched, its class name will be saved to a queue, a first-in-first-out data structure. If another interested activity is launched, its class name will be inserted into the queue after the previous one. Thus, the loading sequence is saved in the queue at runtime. Before exiting the application, the loading sequence will be saved in the ROM of the device through the *SharedPreferences* APIs. When the application launched next time, the preloading module retrieves the loading sequence of its previous usage. After that, the UIs of interested activities are preloaded successively according to it. For example, Figure 8 shows a *SharedPreferences* file saved in the device. The number "0-10" refers to the loading sequence of the interested activities. In this example, the UI of *Activity1* should be preloaded and saved in the view cache first. After it is removed from the cache and loaded to an activity, the UI of *Activity3* will be preloaded according to the sequence.

Figure 8. The loading sequence of a test application.

3.4 Practical Example of Implementation

In this section, I describe how to implement FUEL with the application *InstaMaterial*. In this example, the UI elements of *UserProfileActivity* and *CommentsActivity* need to be preloaded. Their corresponding class names are "*ui.activity.UserProfileActivity*" and "*ui.activity.CommentsActivity*". The resource ID of their UI XML files are *R.layout.activity_user_profile* and *R.layout.activity_comments*.

The implementation steps are shown as follows:

1. Copy the source code files of FUEL and paste to the folder that contains java files, i.e. to app/src/main/java/io/github/froger/instamaterial in this example. Figure 9 shows the hierarchy of project files.

Figure 9. Project files of *InstaMaterial*.

2. Open the application java class file, i.e. *InstaMaterialApplication* in Figure 9, and declare a local variable in that class as follows:

```
public static Preloader preloader = new Preloader();
```

Figure 10. Code of step 2.

3. Override the *onCreate()* method of the *InstaMaterialApplication* class to initialize FUEL (Figure 11).
    a. Save required information through *setPreloadInfo(String name, int id)* method, which includes the activities' class name and the resource ID of UI layout files.
    b. Read the *SharedPreferences* file to get the loading sequence of interesting activities.
    c. Registers an *ActivityLifecycleCallbacks* to monitor the lifecycle of activities.
    d. Start the preloading module with prediction.

```
@Override
public void onCreate() {
    super.onCreate();
    Timber.plant(new Timber.DebugTree());

    //Put required information
    preloader.setPreloadInfo( name: "ui.activity.UserProfileActivity", R.layout.activity_user_profile);
    preloader.setPreloadInfo( name: "ui.activity.CommentsActivity", R.layout.activity_comments);

    //Read loading sequence
    SharedPreferences sharedPreferences = getSharedPreferences( name: "Sequence", MODE_PRIVATE);
    preloader.loadSequence(sharedPreferences);

    //
    registerActivityLifecycleCallbacks(new ActivityLifecycleCallbacks() {...});

    //Start
    preloader.startInSequence();

}
```

Figure 11. Override the *onCreate()* method of the *InstaMaterialApplication* class.

4. Retrieve LayoutInflater from the *context* of the main activity with *setInflater(Context context)* method. That is to find *onCreate()* method of *MainActivity* and add the following code in Figure 12.

```
InstaMaterialApplication.preloader.setInflater(this);
```

Figure 12. Retrieve LayoutInflater.

5. Override the *onDestroy()* method of *MainActivity* with the following code in Figure 13 to enable the loading sequence can be saved in the ROM before exiting the application.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    SharedPreferences sharedPreferences = getSharedPreferences( name: "Sequence", MODE_PRIVATE);
    InstaMaterialApplication.preloader.saveSequence(sharedPreferences);
}
```

Figure 13. Override the *onDestroy()* method of the main activity.

6. Change the UI-APIs in *UserProfileActivity* and *CommentsActivity* with the following code in Figure 14. Note, for *CommentsActivity*, the 7th line should be modified to *setContentView(R.layout.activity_comments)*.

```
Log.d( tag: "MyTest",  msg: "Set Preloaded View start:");
View getView = ViewCache.get(this.getLocalClassName());
if (getView != null) {
    setContentView(getView);
    Log.d( tag: "MyTest",  msg: "Set finish(success)");
} else {
    setContentView(R.layout.activity_user_profile);
    Log.d( tag: "MyTest",  msg: "Set finish(fail)");
}
```

Figure 14. UI operations in *UserProfileActivity* and *CommentsActivity*.

Note that above steps may be slightly different for other Android applications.

Chapter 4. Performance and Evaluation

We have tested FUEL on four applications that are all open source and available in the Google Play Store or on GitHub. All the applications are tested with Samsung Galaxy S9 (SM-G9600) smartphone whose OS version is Android 10.0 (API level 29). The commit number refers to the master version. Table 1 shows the basic information of all tested applications.

App Name	Downloads	Commit #	Source Code Link
Images to PDF	10K+	7f678a0	https://github.com/Swati4star/Images-to-PDF
AntennaPod	500K+	3d7f937	https://github.com/AntennaPod/AntennaPod
Your Master Clean		e57c884	https://github.com/joyoyao/superCleanMaster
InstaMaterial		6a8f626	https://github.com/frogermcs/InstaMaterial

Table 1. Applications tested with FUEL.

Table 2 shows the execution time of tested applications with and without FUEL. The label UI-API refers to the API which is tested with FUEL. For example, *CropImageActivity.setContentView* stands for the *setContentView()* API of *CropImageActivity*. Each UI-API is tested with cold start in both cases. The complexity

measures how complex a UI is. It is defined as the sum of inflation time of all view groups of the UI. For example, Figure 15 shows the UI layout of *CommentsActivity* of *InstaMaterial*. It consists of three view groups: A Toolbar (in the red box), a FrameLayout (in the yellow box) which contains a RecyclerView and a View, a LinearLayout (in the green box) which contains an EditText and a SendCommentButton. Their inflation time is 1.3, 1.9 and 19.7 respectively. Thus, the complexity of this UI is 22.9 in total. As summarized in Table 2, all the UI-APIs' execution time has been reduced with FUEL. The pre-execution allows the device to decrease the response time by up to 65%. More execution time is saved by FUEL for a complex UI, such as *AntennaPod* vs. *You Master Clean*.

App Name	UI-API	Complexity	Execution Time (ms)		
			App with FUEL	Original	Reduction
PDF CONVERTER: Files to PDF	CropImageActivity.setContentView	20.3	10.1	29.2	19.1 (65.4%)
AntennaPod	WidgetConfigActivity.setContentView	31.8	18.1	50.7	32.6 (64.3%)
Your Master Clean	ChangeLanguage.setContentView	13.1	20.8	29.2	8.4 (28.8%)
InstaMaterial	CommentsActivity.setContentView	22.9	26.4	50.1	23.7 (47.3%)
	UserProfileActivity.setContentView	30.1	27	57.5	30.5 (53%)

Table 2. Execution time of tested UI-APIs.

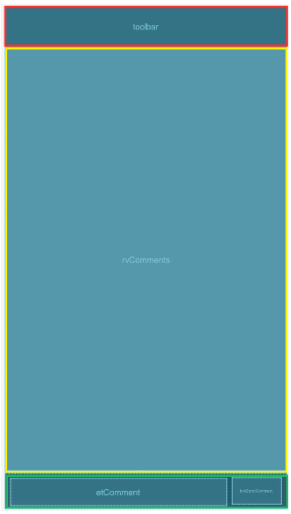


Figure 15. UI layout of *CommentsActivity*.

Table 3 shows the performance of the prediction module. Its data shows the memory space of the inflated view variables in the view cache with and without prediction solution. For the tested applications (i.e. *Image to PDF*, *InstaMaterial* and *UI Test*), each of them needs to preload more than one UI. Specifically, *InstaMaterial* preloads the UIs of *UserProfileActivity* and *CommentsActivity*. *Image to PDF* preloads the UIs of *CropImageActivity* and *ImagesPreviewActivity*. *UI Test* preloads the UIs of *Activity1*, *Activity2* and *Activity3*. If FUEL runs without prediction, all of them are preloaded at runtime and saved in cache. Thus, it takes more storage space. On the contrary, there is only one UI layout saved in cache when FUEL runs with prediction. As shown in Table 3, the memory overhead is decreased by prediction solution.

App Name	Memory of Preloaded UI (Byte)		
	Without Prediction	With Prediction	Memory Saved
Images to PDF	49331	35989	13342 (27.0%)
InstaMaterial	163610	84190	79420 (48.5%)
UI Test	167466	108040	59426 (35.5%)

Table 3. Comparison of memory usage.

Heavy UI rendering may cause soft hangs when the response time is longer than a perceivable delay of 100ms. Table 4 shows the execution time of *CommentsActivity.setContentView* on a virtual device (i.e., the emulator provided by Android Studio) which includes soft hang occurrence. As shown in Table 3, FUEL can reduce the execution time to less 100ms and make the UI-APIs responsive.

	Execution Time (ms)	
	App with FUEL	Original
1	81	101
2	65	127
3	73	95
4	87	97
5	73	113
6	75	105
7	38	116
8	68	75
9	79	99
10	79	102
11	76	64
12	78	123
13	71	112
14	63	85
15	88	131
16	64	95
17	68	130
18	65	85
19	71	84
20	59	106
average	71.05	102.25

Table 4. Execution time of *CommentsActivity.setContentView* on a virtual device.

Chapter 5. Conclusion

This paper presents FUEL, a runtime methodology that preloads the UI elements that are likely to be executed in order to reduce the response time. FUEL consist of three components: the view cache, the preloading module and the prediction module. It records the loading sequence of activities with complex UI at runtime and predicts which UI layout should be inflated ahead of time. After inflation, the generated views are saved in the view cache and can be retrieved through its corresponding class name when needed. In this paper, I test FUEL with four open source applications. The results show that FUEL is able to reduce the execution time of UI-APIs by at least 28%, thus to eliminate soft hangs caused by complex UIs.

References

[1] Brocanelli, Marco & Wang, Xiaorui. (2018). Hang doctor: runtime detection and diagnosis of soft hangs for smartphone apps. 1-15. 10.1145/3190508.3190525.

[2] Brad Fitzpatrick. 2010. Writing zippy Android apps. In Google I/O Developers Conference.

[3] https://developer.android.com/guide/platform.

[4] https://developer.android.com/guide/components/activities/tasks-and-back-stack.

[5] https://developer.android.com/reference/android/app/Activity.

[6] https://developer.android.com/guide/topics/ui/declaring-layout.

[7] Lin, Y.R. & Chu, Edward & Chang, Evan & Lai, Yuan-Cheng. (2017). Smoothed Graphic User Interaction on Smartphones with Motion Prediction. IEEE Transactions on Systems, Man, and Cybernetics: Systems. PP. 1-13. 10.1109/TSMC.2017.2685243.

[8] Linares-Vásquez, Mario & Bavota, Gabriele & Bernal-Cárdenas, Carlos & Oliveto, Rocco & Di Penta, Massimiliano & Poshyvanyk, Denys. (2014). Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study. 11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings. 10.1145/2597073.2597085.

[9] https://blog.csdn.net/nugongahou110/article/details/49662211.

Appendix A. Source Code

A.1 ViewCache.java

```
import android.view.View;

import java.util.HashMap;

public class ViewCache {

   private static final int maxSize = 10;
   private static int nViews = 0;
   private static HashMap<String, View> mCache = new HashMap<>();

   public static synchronized void put(String name, View view) {
      if (nViews == maxSize) {
         return;
      }
      mCache.put(name, view);
      nViews++;
   }

   public static synchronized View get(String name) {
      if (nViews == 0) {
         return null;
      }
      if (mCache.containsKey(name)) {
         nViews--;
         return mCache.remove(name);
      } else {
         return null;
      }
   }

   public static boolean isEmpty(){
      return (nViews == 0);
   }

   public static boolean contains(String name) {
      return mCache.containsKey(name);
   }

}
```

A.2 Preloader.java

```
import android.content.Context;
import android.content.SharedPreferences;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Queue;
import java.util.Set;

public class Preloader {

    private final String TAG = "MyTest";
    private LayoutInflater mInflater;
    private boolean stop;
    private ArrayList<PreloadInfo> mPreloadInfo = new ArrayList<>();
    private Queue<String> sequence = new LinkedList<String>();
    private Queue<String> sequenceRecord = new LinkedList<String>();

    public String currentActivity;
    public String previousActivity;

    public void start() {
        Log.d(TAG, "Preloader start");
        stop = false;
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (!stop) {
                    if (needPreload() && mInflater != null) {
                        for (int i=0; i < mPreloadInfo.size(); i++) {
                            PreloadInfo info = (PreloadInfo) mPreloadInfo.get(i);
                            if (!ViewCache.contains(info.Name)) {
                                View preloadView = mInflater.inflate(info.Id, null);
                                ViewCache.put(info.Name, preloadView);
                                //Log.d(TAG, "View saved in cache: " + info.Name);
                            }
```

```
                }
            }
        }
    }
}).start();
}

public void startInSequence() {
    Log.d(TAG, "Preloader start in sequence");
    stop = false;
    /*for (String str:sequence) {
        Log.d(TAG, str);
    }*/

    new Thread(new Runnable() {
        @Override
        public void run() {
            while (!stop) {
                if (ViewCache.isEmpty() && mInflater != null){
                    String name = sequence.poll();
                    for (int i=0; i < mPreloadInfo.size(); i++) {
                        PreloadInfo info = (PreloadInfo) mPreloadInfo.get(i);
                        if (info.Name.equals(name)) {
                            View preloadView = mInflater.inflate(info.Id, null);
                            ViewCache.put(info.Name, preloadView);
                            //Log.d(TAG, "View saved in cache: " + info.Name);
                            break;
                        }
                    }
                }
            }
        }
    }).start();
}

public void setPreloadInfo(String name, int id) {
    PreloadInfo info = new PreloadInfo(name, id);
    mPreloadInfo.add(info);
}

private boolean needPreload() {
    for (int i=0; i < mPreloadInfo.size(); i++) {
        PreloadInfo info = (PreloadInfo) mPreloadInfo.get(i);
        if (!ViewCache.contains(info.Name)) {
```

```
            return true;
        }
    }
    return false;
}

public boolean containsInfo(String name) {
    for (int i=0; i < mPreloadInfo.size(); i++) {
        PreloadInfo info = (PreloadInfo) mPreloadInfo.get(i);
        if (info.Name.equals(name)) {
            return true;
        }
    }
    return false;
}

public void recordSequence(String name) {
    sequenceRecord.offer(name);
}

public void setInflater(Context context) {
    mInflater = LayoutInflater.from(context);
}

public void saveSequence(SharedPreferences sharedPreferences) {
    SharedPreferences.Editor editor = sharedPreferences.edit();
    int n = 1;
    while (!sequenceRecord.isEmpty()) {
        String name = sequenceRecord.poll();
        String number = Integer.toString(n);
        editor.putString(number, name);
        n++;
    }
    Log.d(TAG, "Sequence saved");
    editor.apply();
}

public void loadSequence(SharedPreferences sharedPreferences) {
    Map<String, ?> all = sharedPreferences.getAll();
    if (!all.isEmpty()) {
        Set<String> keys = all.keySet();
        int n = 1;
        while (keys.contains(Integer.toString(n))){
            //Log.d(TAG, key);
```

```
            String key = Integer.toString(n);
            String name = (String) all.get(key);
            sequence.offer(name);
            n++;
         }
      }
      sharedPreferences.edit().clear().apply();
      Log.d(TAG, "Sequence loaded");
   }

   public int getInfoId(String name) {
      for (int i=0; i < mPreloadInfo.size(); i++) {
         PreloadInfo info = (PreloadInfo) mPreloadInfo.get(i);
         if (info.Name.equals(name)) {
            return info.Id;
         }
      }
      return 0;
   }

   public void stop() {
      stop = true;
   }

}

class PreloadInfo {
   public String Name;
   public int Id;
   public LayoutInflater inflater;

   public PreloadInfo(String name, int id) {
      Name = name;
      Id = id;
      inflater = null;
   }

   public void setInflater(Context context) {
      inflater = LayoutInflater.from(context);
   }
}
```

A.3 Register an ActivityLifecycleCallbacks

```
registerActivityLifecycleCallbacks(new ActivityLifecycleCallbacks() {
   @Override
   public void onActivityCreated(@NonNull Activity activity, @Nullable Bundle
savedInstanceState) {
      //Log.d(TAG,"onActivityCreated: " + activity.getLocalClassName());
      String name = activity.getLocalClassName();
      if (preloader.containsInfo(name)) {
         preloader.recordSequence(name);
      }
   }

   @Override
   public void onActivityStarted(@NonNull Activity activity) {
      //Log.d(TAG,"onActivityStarted: " + activity.getLocalClassName());
   }

   @Override
   public void onActivityResumed(@NonNull Activity activity) {
      //Log.d(TAG, "onActivityResumed: " + activity.getLocalClassName());
   }

   @Override
   public void onActivityPaused(@NonNull Activity activity) {
      //Log.d(TAG,"onActivityPaused: " + activity.getLocalClassName());
   }

   @Override
   public void onActivityStopped(@NonNull Activity activity) {
      //Log.d(TAG, "onActivityStopped: " + activity.getLocalClassName());
      //previousActivity = activity.getLocalClassName();
   }

   @Override
   public void onActivitySaveInstanceState(@NonNull Activity activity, @NonNull
Bundle outState) {
   }

   @Override
   public void onActivityDestroyed(@NonNull Activity activity) {
      //Log.d(TAG,"onActivityDestroyed: " + activity.getLocalClassName());
   }
});
```