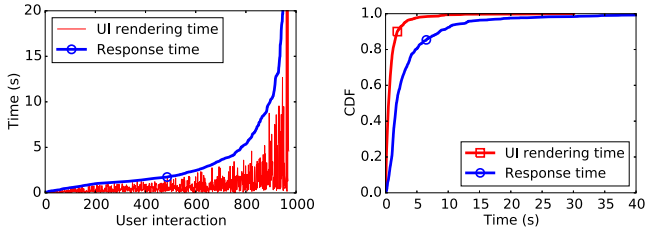
Every Pixel Counts: Fine-Grained UI Rendering Analysis for Mobile Applications

Yi Gao[1], Yang Luo[1], Daqing Chen[2], Haocheng Huang[1], Wei Dong[1*] Mingyuan Xia[3], Xue Liu[3], Jiajun Bu[1]
[1]College of Computer Science, Zhejiang University, China.
[1]Zhejiang Provincial Key Laboratory of Service Robot, China.
[2]China Mobile, China.
[3]School of Computer Science, McGill University, Canada.
Email: {*gaoyi, luoy, huanghc, dongw, bjj*}*@zju.edu.cn*, *chendaqing@chinamobile.com*,
xueliu@cs.mcgill.ca, *mingyuan.xia@mail.mcgill.ca*

***Abstract*—For mobile apps, user-perceived delays are critical for user satisfaction. According to our measurement, long delays are commonly caused by network and storage I/O operations while short delays are mainly caused by UI rendering. Short delays are not uncommon, which account for 55.3% in our measurement cases. Previous app performance studies have largely focused on I/O operations but the understanding of UI rendering impact is limited. In this work, we propose DRAW, a system that performs two UI rendering analyses to help app developers pinpoint rendering problems and resolve short delays. The first analysis outlines the wasted rendering time on invisible or covered UI components, namely the overdraw problem. The second analysis is to identify the responsible UI components and rendering operations that cause overall low rendering efficiency. We implement DRAW on Android and apply it to study 1,158 real-world Android apps. Results show that DRAW is helpful as it can pinpoint the responsible UI components and specific rendering operations. Four concrete case studies of real-world apps are further presented to show how DRAW can help developers improve the UI rendering performance of their apps.**

I. Introduction

The past decade has witnessed a tremendous growth of the number of smartphones and mobile applications [22]. More than one billion smartphones are shipped in a single year [5], and more than three million mobile applications are available in online application markets [2]. Smartphones and mobile applications have been playing more and more important roles in our daily lives. According to a survey conducted by Tecmark [3], a user carries out 221 tasks per day on his/her smartphone on average, and the average usage time is longer than three hours a day. Therefore, it is crucial to study the performance of mobile applications. In addition, a recent study shows that 50% of mobile phone user interactions last less than 30 seconds [23]. With such short user interactions, it is important that mobile applications keep responsive.



(a) distribution (b) cdf

Fig. 1. UI rendering time v.s. total response time.

Due to the importance of user responsiveness, there have been many studies focusing on measuring and optimizing responsiveness of mobile applications. In particular, intensive research efforts have focused on the operations which often cause excessively long delays, such as network [21], [23], [18] and storage [16], [12] I/Os. Although these operations are indeed very important to the application responsiveness, we observe that UI rendering performance also has significant impact on application responsiveness. In order to quantify the importance of UI rendering operations, we conducted a measurement study with about 1,000 user interactions (e.g., click, scroll) using 30 popular mobile applications. During the experiments, we measure the rendering time and the total response time. Then we report the UI rendering time and the total response time in Figure 1, sorted by the total response time. We draw the following two observations. First, about 20% of user interactions take longer than five seconds. After inspecting the log, we found out that these excessively long delays are caused by network and storage I/Os, which have been well studied in existing work [20], [8], [17], [12], [16]. However, for short response times (<2s), the UI rendering time accounts for the majority (49% to 97.8%). These short user interactions accounts for 55.3% of all user interactions, which are the main target of this work.

Our goal is to develop a system that analyzes the UI rendering performance of mobile applications. Given an app binary, the system should be able to automatically generate a report to help the developer improve the rendering performance of the app in the quality assurance process. We present DRAW, a UI rendering analysis tool for mobile

This work is supported by the National Science Foundation of China (No. 61472360 and No. 61502417), Zhejiang Provincial Natural Science Foundation of China (No. LY16F020006), Zhejiang Provincial Key Research and Development Program (No. 2017C02044), CCF-Tencent Open Research Fund, Fundamental Research Funds for the Central Universities (2016QNA5012, 2016FZA5010), and China Ministry of Education−China Mobile Joint Project (No. MCM20150401). *Wei Dong is the corresponding author.

applications, to meet the design goal. Given an app binary, DRAW runs the app and performs fine-grained tracing of UI rendering operations (Section III). Then based on the traced data, DRAW performs overdraw analysis (Section IV) and rendering performance diagnosis (Section V). **1) Overdraw analysis.** In this paper, overdraw includes two kinds of drawing behaviors, namely repeated-draw and invisible-draw. Repeated-draw is very common in modern app designs with stacking and layering, in which a pixel may be drawn more than once in the same frame. When repeated-draw happens, only the last drawing is visible and the previous drawing operations are wasted, degrading rendering performance. Invisible-draw means a view object is rendered but not visible to user, usually caused by view draw out of screen or being set to be invisible by developers. We define a *rendering efficiency score* for each view object (Android window UI component). The overdraw analysis outputs an ordered list of view objects according to their scores. Further, the overdraw analysis also analyzes the overdraw features on a user interaction basis, which could identify the rendering problems in the code path responding to user inputs. For each user interaction started with a user input, several overdraw-related features are extracted for further performance diagnosis. **2) Rendering performance diagnosis.** UI rendering is a complicated process, and its performance is related to many internal rendering features, including overdraw-related features. DRAW performs correlation analysis and causality analysis to find the features causing the observed performance issues and pinpoints the suspicious view objects.

We implement DRAW and use it to analyze UI rendering performance of 1,158 mobile applications. In total, 10,966 user interactions with rendering operations of 307,048 view objects are analyzed by DRAW. Compared with the default overdraw analysis tool provided by the Android system, DRAW pinpoints the suspicious view objects and the corresponding rendering operations more specifically for problem fixing. Further, we present four concrete case studies to show how DRAW can help developers improve the UI rendering performance of their apps. According to the analysis results of DRAW, we modify (decompile, modify and rebuild) two apps used in the case study and evaluate the UI rendering performance. Results show that the number of dropped frames is decreased by 16.5% and 26.9% in these two apps. We believe that with app source code in hand, developers can further improve the UI rendering performance of their apps using the analysis results generated by DRAW.

The rest of this paper is organized as follows. Section II gives an overview of DRAW. Section III presents the measurement framework of DRAW. Section IV and Section V describe the overdraw analysis and performance diagnosis in detail. Section VI shows the evaluation results of DRAW. Section VII discusses the related work, and finally, Section VIII concludes this paper.

II. Overview

Figure 2 gives an overview of using DRAW to analyze UI rendering performance of mobile applications. After an application binary is submitted to DRAW, it is pushed to testing devices with automated tests. A *UI automation engine* [4] is used to generate user inputs automatically. When the application is running, the *rendering performance measurement* component of DRAW records fine-grained information about the rendering process. Then a *trace preprocessing component* of DRAW preprocesses the raw data and performs overlapped rendering time elimination and source code mapping. The output of the preprocessing is the *DRAW-trace* for rendering performance analysis. The DRAW-trace includes a *rendering-log* and a *system-events-log*. The rendering-log includes the execution times of various rendering operations for all view objects (e.g., a text box or a button). The system-event-log includes a sequence of user interactions with timestamps, which is used to generate analysis units for user-interaction-oriented performance diagnosis.

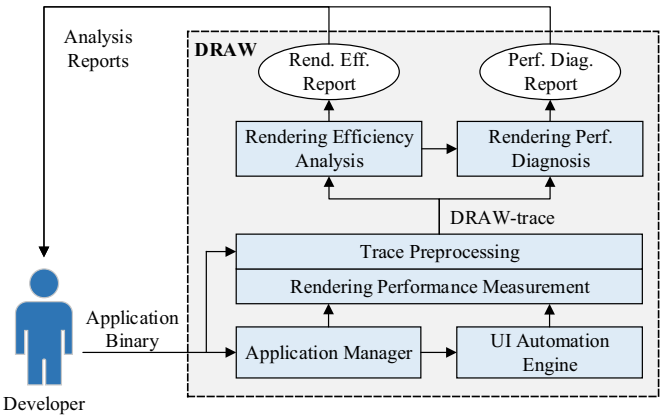


Fig. 2. An overview of DRAW.

The *rendering efficiency analysis* component generates two kinds of outputs, an overdraw report and overdraw-related feature data for rendering performance diagnosis. The overdraw report contains a list of view objects, sorted by the severity of their overdraw problems. The overdraw-related feature data is submitted to the performance diagnosis component for rendering performance diagnosis. The second component is the rendering *performance diagnosis* component, which takes data from both the DRAW-trace and the overdraw analysis component. It outputs a diagnosis report which includes the root cause about the observed rendering performance problem.

III. Measurement Framework

A. Performance Measurement of Rendering Operations

1) Rendering-log: We instrument the Android application framework layer to collect rendering-related log for running apps. Our log is a chronologically sequence of rendering operations, each line describing one operation. In Android, each rendering operation is associated with one particular UI component (namely `View`).

View Properties. Each view has a unique identifier (`vid`) obtained by calling `System.identityHashCode(obj)` in Java. For a view, we collect various properties that are related to rendering, including its size (`width` and `height`), screen position (`left` and `top`), and background transparency (`hasbackground`). All sizes and positioning are in terms of pixels. In addition to the runtime size (the size of the UI component at runtime), we also log *static* size of a view to understand how a view object inflates during runtime. Static sizes can be obtained by parsing the UI design XML files in the app.

View Hierarchy. In Android, views have "containing" relationship. Essentially, there are views that have concrete content (e.g. a button, text box or an image box) as well as views (e.g. a grid, a list) that are totally transparent and only regulate the placement of its containing views. Thus, all UI components on the screen have a hierarchical relationship (a multi-leaf tree). We capture this by recording the `vid` of the parent view and reconstruct the hierarchy structure during offline analysis.

Timing and Operation Type. This information helps us to identify time-consuming operations and debug bottlenecks in rendering. For timing, we record the start time and duration of the operation, in nanoseconds, with the help of a high definition clock source in Android system (`SystemClock.elapsedRealtimeNanos()`). To further understand if the timing would affect responsiveness, we also record whether the operation happens on the UI thread.

The entire rendering process mainly includes three types of operations. First, the *measure* operation determines the size (i.e., height and width) of the view to be rendered. A view might have a developer-defined size, inherit size from its parent view (`match_parent`) or have a large enough size of wrap its content (`wrap_content`). During the measure operations, `match_parent` and `wrap_content` sizes are converted to actually pixel numbers. Then, the *layout* operation decides where to place the view. This is computed by different placement strategies. For example, the simplest strategy is to stack all views vertically (known as the `LinearLayout`). Thus the top of the first view is 0, the top of the second one is the height of the first view, so on and so forth. Finally the *draw* operation actually paints the view content on a canvas. The *draw* operation performance is heavily affected by GPU acceleration. Thus we also record if the view is hardware accelerated. To collect all these view-related information, our modification to the Android application framework layer mainly resides in the base class for Views, i.e., `android.view.View`.

Since all the modifications are made within the Android application framework layer, a developer can easily use DRAW by replacing the `framework.jar` without modifying the low-level parts of the Android system.

There is a tradeoff between the scale of the collected data and the effectiveness of the performance analysis. In Android, it is possible to collect more information by modifying the framework layer and lower layers [24], such as all function calls and inter-process communications. By analyzing these additional information, we may be able to obtain more accurate and fine-grained analysis results. However, collecting such information could introduce non-negligible overhead, e.g., up to 300MB network IO per day in Panappticon [24]. Therefore, in order to perform fine-grained UI rendering performance analysis without introducing significant overhead, we only collect information closely related to the drawing operations of view objects in the design of DRAW.

2) System-events-log: In addition to per-view information, we also collect two kinds of system-wise events, user inputs and frame rate (FPS). User inputs are used to generate rendering analysis unit for performance diagnosis. In DRAW, we instrument two framework classes, `android.view.GestureDetector` and `android.view.View` to track user inputs, e.g., drag, click, long click, and scroll. FPS is used as a high-level metric for the smoothness of UI rendering. Concretely, we use the `atrace` tool in Android to trace the frame rate in a fine-grained manner.

B. Trace Preprocessing

The trace preprocessing includes two main parts, overlapped rendering time elimination and source code mapping.

Overlapped render time elimination. In Android, views have hierarchical relationship. Each view have two sources of overlapped rendering time. First, the rendering process of a parent view object includes all of its children view objects. Therefore, the recorded execution times are overlapped to each other. Second, the three rendering operations (i.e., `measure, layout, draw`) are not strictly sequentially executed. For example, after a *measure* operation, the *measure* operation may be called again from a *layout* operation. This causes the execution times of these rendering operations to be also overlapped.

In order to address the above execution time overlapping problem, DRAW goes through the view hierarchy and updates the execution time of each view object recursively. Then for all rendering operations of each view object, DRAW ensures that there is no overlapped operation. After the preprocessing, the execution time of each view object reflects the actual time used to render itself, providing accurate performance measurement data for further analysis.

Source code mapping. The recorded view id is only a hashed id of that view object at runtime. In order to help the developer locate the exact places that view object is defined and used, DRAW performs source code mapping as follows.

Figure 3 shows the source code mapping method used in DRAW. DRAW first uses the `aapt` tool to map the recorded view id (e.g., `0x7f0e0390`) to its XML id name (e.g., `activepager1`). Then the XML id name can be mapped to the layout file (where the view is defined, e.g., `layoutsign_up_activity_common.xml`) through name matching. Finally the exact location where this view object is loaded (e.g., `LandingScreenActivity.java`) can be recovered by searching the use of this layout file in the Java code. In practice, this method can recover about $2/3$ of the recorded view ids. Remaining views are created dynamically in the code and thus the XML id is not available. Therefore, we use a view hierarchy matching method to locate the view for the app developers. Concretely, each view has a certain position in the view hierarchy. We use this position of a certain view as its unique feature and perform a matching in all views.

IV. Fine-grained Overdraw Analysis

As mentioned in the introduction section, DRAW focuses on two overdraw behaviors, invisible-draw and repeated-draw. Invisible-draw happens when a view object is rendered but not visible to users. In most cases, this happens when the view object is out of the screen or app developer manually sets it to be *invisible*. However, an invisible view object may waste resources for rendering.

Another overdraw behavior is repeated-draw. In order to enable more beautiful designs of mobile applications,

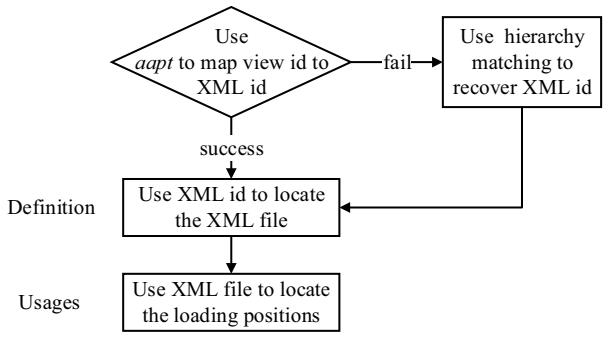


Fig. 3. Source code mapping of view definition and usages.

modern layouts (e.g., the Android rendering system) usually provide stacking and layering functionalities to developers. However, inappropriate use of stacking and layering may cause performance problems. When a pixel is drawn more than once in the same frame, only the last drawing is visible, which is called *repeated-draw* in this paper.

A. Overdraw Analysis in DRAW

DRAW takes the rendering-log as input, and the output includes an overdraw analysis report and other results for further performance diagnosis. A rendering-log includes a list of UI rendering operation records sorted by starting time. After processing the rendering-log, the output of overdraw analysis includes two parts, an overdraw analysis report and overdraw features for further rendering performance diagnosis.

Overdraw analysis report. The overdraw analysis report is an ordered list of view objects. A view object with higher rank in the list indicates more severe overdraw problem of that view object. We define *rendering efficiency score* (RES) of a view object v_i by the following equation.

$$RES(v_i) = \frac{\sum_k U(v_i^{(k)}) + 1}{\sum_i C(v_i^{(k)})}, \quad (1)$$

where $U(v_i^{(k)})$ and $C(v_i^{(k)})$ are the utility and cost of rendering the k^{th} occurrence of view object v_i in the entire trace, respectively. The utility of a view object depends on the number of its *visible* pixels and the length of time when these pixels are visible. Concretely, the following equation gives the definition of the $U(v_i^{(k)})$.

$$U(v_i^{(k)}) = \sum_{px_j \in v_i^{(k)}} t(px_j, v_i^{(k)}), \quad (2)$$

where px_j represents each pixel in $v_i^{(k)}$ and $t(px_j, v_i^{(k)})$ is the visible time of that pixel of $v_i^{(k)}$. Given a rendering-log which includes all rendering operations of all view objects with timing information, DRAW calculates the utility of each view object. The cost of rendering a view object is defined as the length of time (in seconds) used to render it, including the `measure`, `layout` and `draw` operations. Since a view object could be completely invisible all the time, its utility could be zero. In order to differentiate zero utility view objects with different costs, there is a "plus 1" in Equation 1.

This definition of RES considers both the repeated-draw behavior and the invisible-draw behavior at the same time. In addition, it captures the dynamic overdraw behavior by considering fine-grained timing information, providing an estimation of the rendering efficiency of each view object in term of overdraw.

TABLE I. OVERDRAW-RELATED FEATURES

Feature Type	Feature Description
maximum overdraw depth	maximum number of times a pixel is drawn in the same frame
average overdraw depth	average number of times a pixel is drawn in the same frame
number of invisible view objects	number of invisible view objects during the whole user interaction
average effective drawing ratio	average ratio of total changed visible pixels and total drawn pixels for all frames with draw operations

Overdraw features for further responsiveness diagnosis. In the design of DRAW, overdraw analysis not only outputs an overdraw analysis report, but also generates overdraw features for further responsiveness diagnosis. In responsiveness diagnosis, the analysis unit is each *user interaction* started from a user input. Given a rendering-log and a system-event-log (including a sequence of timestamps indicating multiple user interactions), DRAW will generate multiple overdraw-related features for each analysis unit, i.e., user interaction.

Table I gives a list of these features. The first two features are about the overdraw depth. During the rendering process of a frame, the overdraw depth of a pixel is the number of drawings of that pixel. Since redrawing the same pixel multiple times in the same frame is completely a waste of time, the DRAW extracts these two pixel-depth-related features for diagnosis. The third feature is the number of view objects which are completely invisible during the whole user interaction. The last overdraw-related feature is the average effective drawing ratio. The calculation of this feature is as follows. For each frame with `draw` operations, DRAW calculates two values, the number of changed pixels and the number of total drawn pixels. Due to the existence of overdraw, the number of changed pixels is always smaller than or equal to the number of total drawn pixels. Then DRAW calculates an average value of the ratio of these two values, which is the fourth overdraw-related feature.

V. UI RENDERING PERFORMANCE DIAGNOSIS

In this paper, we focus on UI rendering performance in terms of responsiveness. We use the *rendering time* trigger by a user input and the *average frame rate* as two key performance metrics.

A. Performance Metrics and Rendering Features

In DRAW, a user interaction starts from a user input, and ends when the rendering operations related to this input are completed. In each experiment, a sequence of user inputs are recorded. The start time of each user interaction is the start time of each user input. DRAW determines the end time of each user interaction as the completion time of the last rendering operation before the next user input. The first performance metric, rendering time, can be calculated by adding the durations of all rendering operations of that user

TABLE II. RENDERING FEATURES FOR DIAGNOSIS

Feature Type	Feature Description
overdraw-related features	features described in Table I
statistical features of rendering operation execution time	average/maximum/minimum/variance of measure/layout/draw operation execution times of all view objects
statistical features of rendering operation execution number	average/maximum/minimum/variance of measure/layout/draw operation execution number of all view objects
number of view objects	number of view objects drawn during the user interaction
statistical features of pixel depth	average/maximum/minimum/variance of all pixel depths
statistical features of view object area	average/maximum/minimum/variance of view object area
features of view tree	maximum depth and average depth of the view tree
rendering time ratio	ratio of rendering time and total user interaction time
binary features	binary features like "whether hardware acceleration is enabled"

interaction. The second performance metric is the average frame rate during a user interaction.

A number of rendering features are shown in Table II, including the overdraw-related features. In addition to some simple features like "number of view objects", there are also statistical features which are average/maximum/minimum/variance of a certain characteristic of view objects. For example, a typical statistical feature could be "maximum execution time of draw operation". In total, more than 40 features are used in the current design of DRAW.

B. Performance Diagnosis

Similar to the overdraw analysis, the output of the performance diagnosis is a number of view objects which may be the root causes of the observed performance issues. Before describing the detailed diagnosis algorithm, we first introduce the following notations.

- $f_1, f_2, ..., f_N$, N rendering features;
- p_1, p_2, two performance metrics, p_1 represents the rendering time and p_2 represents the average frame rate;
- $corr(f_i, f_j)$ and $corr(f_i, p_j)$, *correlation coefficient* between a feature f_i and another feature f_j or a performance metric p_j;
- $v_1, v_2, ..., v_M$, M view objects;
- $sig(f_i, p_j)$, *significance score* of a feature f_i to a performance metric p_j;
- $con(v_i, f_i)$, *contribution* of a view object v_i to a feature f_j.

In order to quantify the significance score for each feature, we construct a *correlation graph* CG_i for each performance metric p_i. The diagnosis for the two performance metrics are two separate processes. Therefore, in the following description, we use the rendering time as the performance metric p. A *CG* is a directed acyclic graph (DAG). The nodes of the graph *CG* are all the rendering features and the performance metric.

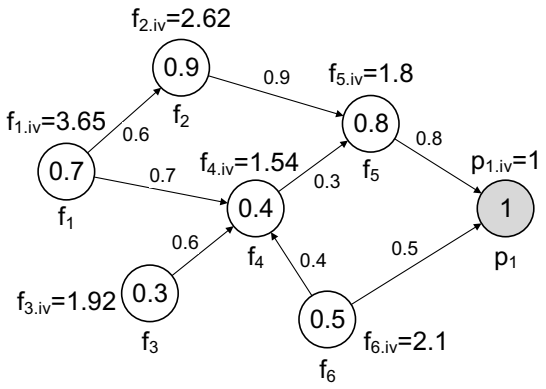


Fig. 4. An example of correlation graph. The *iv* value calculation of each feature is given in Algorithm 1.

Algorithm 1 The significance score calculation algorithm

Input: A correlation graph *CG*
Output: Significance score $sig(f_i, p)$ of each feature f_i

```
1:  procedure SIGNIFICANCE-SCORING
2:      Let p (performance metric) be the root node of CG
3:      p.iv = 1 // an intermediate value
4:      Let Queue be an empty FIFO queue of nodes
5:      Queue.Enqueue(p)
6:      while Queue is not empty do
7:          f = Queue.Dequeue()
8:          if f has parent nodes in Queue then
9:              Queue.Enqueue(f)
10:         else
11:             f.iv = 1 // an intermediate value of f
12:             for each parent node r of f do
13:                 f.iv = f.iv + r.iv × w_fr
14:             for each child node c of f do
15:                 Queue.Enqueue(c)
16:     Let max(iv) be the maximum value of all f.iv
17:     for each node f in CG do
18:         sig(f, p) = corr(f, p) + f.iv / max(iv)
```

Each node i (i.e., feature f_i) is associated with a value which is the correlation coefficient between the feature f_i and the performance metric p, i.e., $corr(f_i, p)$. The directed links of the graph represent the dependency between pairs of nodes, which are determined by calculating a Bayesian network of these features and the performance metric p. Then the weight w_{ij} of a link l_{ij} from a feature f_i to another feature f_j is the correlation coefficient between f_i and f_j, i.e., $corr(f_i, f_j)$. Figure 4 shows an example of such a correlation graph. Six features and a performance metric are represented by seven nodes in the graph whose structure is obtained by a Bayesian network learning process. For example, the value of f_2 is $corr(f_2, p_1) = 0.9$, which means that the correlation coefficient between feature f_2 and the performance metric p_1 is 0.9. The weight w_{14} is $corr(f_1, f_2) = 0.7$, which means the correlation coefficient between the two features f_1, f_2 is 0.7. The figure also shows the *iv* value of feature f, whose calculation will be given in Algorithm 1.

Then based on the correlation graph, we can calculate the *significance score* for each feature. Algorithm 1 describes the calculation of significance score for each feature. The input of the algorithm is the correlation graph and the output is the

TABLE III. APPS USED IN CASE STUDIES

App name	Version	Downloads	Description	# of views
PicsPlay	3.6.1	5M - 10M	an app for photo editing	3719
Pregnancy+	3.3.1	1M - 5M	an app providing professional advices and service for those who has a pregnant family member	4408
FaceQ	3.4.0	10M - 50M	an app for creating cute cartoon avatar	1470

significance score for each feature. There is an intermediate value (i.e., *iv* value) associated with each feature (line 11) and the performance (line 3). The intuition of this intermediate value is that a feature is more likely to be a root cause if it 1) is far away from the node representing the performance metric (e.g., in Figure 4, f_1 is more likely to be the root cause than f_2, since f_2 depends on f_1), and 2) has large correlation coefficients along the path towards the performance metric node. The calculation starts from the node p and conduct a BFS (line 6 to line 15). The intermediate value of each feature is the sum of weighted intermediate values of all its parent nodes (line 13). The weight w_{fr} is the correlation coefficient between the feature f and its parent feature r, i.e., $corr(f,r)$. In the example shown in Figure 4, the *iv* value of each feature is calculated according to this algorithm. For example, $f_1.iv = 1 + f_2.iv \times w_{12} + f_4.iv \times w_{14} = 1 + 2.62 \times 0.6 + 1.54 \times 0.7 = 3.65$. After the BFS, the algorithm calculates the significance score for each feature f based on the following two factors, the correlation between the feature and the performance metric $corr(f,p)$ ($\in [0,1]$), and a normalized intermediate value $\frac{f.iv}{max(iv)}$ ($\in [0,1]$). In the example, $sig(f_1,p) = 0.7 + 1 = 1.7$.

The final step is to use the significance score of each feature to pinpoint a number of view objects which may cause the performance issues. DRAW calculates a *root cause score* for each view object v_i, i.e., $RCS(v_i)$. In order to calculate each $RCS(v_i)$, we further introduce a concept of *contribution* of a view object to a feature. The calculation of contribution depends on the characteristics of different features. For example, for a feature "number of view objects", the contribution of each view object to this feature is obviously the same. However, for a feature "maximum execution time of draw operation", the contribution of each view object with the maximum drawing time is $1/m$, where m is the number of such view objects with maximum drawing time, and the contributions of the rest view objects are all zero. Based on the significance score of each feature and the contribution of each view object, DRAW calculates the *root cause score* for each view object as follows.

$$RCS(v_i) = \sum_{1 \le j \le N} sig(f_j, p) \cdot con(v_i, f_j), \quad (3)$$

where N is the number of features and p is the performance metric (i.e., rendering time). After the *RCS* calculation for all view objects, DRAW output a number of view objects with the largest *RCS* values.

VI. EVALUATION

We collect two sets of mobile applications for evaluating the effectiveness of DRAW. The first set includes 170 popular mobile applications, with an average of 7.5 million downloads. In order to cover more possible user interactions, we use an user interaction recording tool [1] to record a sequence of user interactions and replay the sequence multiple times. Since the user interactions are manually generated, a large number of different user interactions are performed. The measurement study described in the introduction section uses applications in this set. The second set includes 978 applications randomly sampled from online application markets. For these applications, we use an UI automation tool [4] to generate user interactions automatically. Although the automatically generated user interactions cannot achieve the same coverage as the manually generated ones, it is able to generate a large number of user interactions to evaluate the effectiveness of DRAW. In total, 1,158 different mobile applications are analyzed by DRAW.

Overhead. DRAW is implemented in Python (PC side analysis) and Java (phone side measurement). At the phone side, DRAW uses the `logcat` tool in Android to record measurements and sends the measured data to a PC. When a developer is testing an app, on average, DRAW will record 172 lines of logs per second. There is no SD card I/O or network I/O involved in the measurement process. We did not observe performance degradation after DRAW was enabled in the apps tested. Concretely, we recorded the CPU utilization and memory consumption before and after enabling DRAW of all apps tested on different phones. On overage, the memory consumption is increased by 8.13MB and the CPU utilization is increased by less then 2.5% after DRAW is enabled. In these experiments, we also observed that the CPU utilization was decreased after DRAW was enabled on some phones, which was due to the measurement error caused by the indeterministic behaviors of these phones. These results show that the overhead of DRAW is negligible.

In the evaluation, four different phones/tablet are used: one Nexus4, one Nexus5, one Nexus7, and one Nexus10. In the following, we first present some overall statistics about rendering performance of various view types. This could help outline the overall rendering performance of UI widget in Android apps. Then we give the evaluation results of rendering performance diagnosis. More importantly, four case studies will be presented in detail to show how DRAW can effectively help developers improve their rendering performance of their apps. We will also give some implications for improving rendering performance of apps in this section. Table III summarizes the apps used in our case studies.

A. Rendering Performance of Different View Types

Results of Overdraw Analysis. Figure 5 shows the overall results of overdraw analysis. In total, 127,732 view objects are analyzed. 20 columns are shown in the figure. Each column shows the ratios of different view types whose RES are within a certain RES range. For example, the RES of all view objects in the third column (e.g., RES range ID is 3) are in (3.8×10^{-4}, 5.7×10^{-4}). These RES ranges are determined by keeping the number of views in each range be the same. Therefore, each range includes more than 6,000 views in Figure 5. Since the rendering of a view object is more efficient when its RES value is larger, a column with a larger RES range ID includes the view objects with higher rendering efficiency. Therefore, from the figure, we can see that for `ImageView`, its rendering efficiency is high since most of `ImageViews` are in the right part of the figure. Different from `ImageView`, the rendering

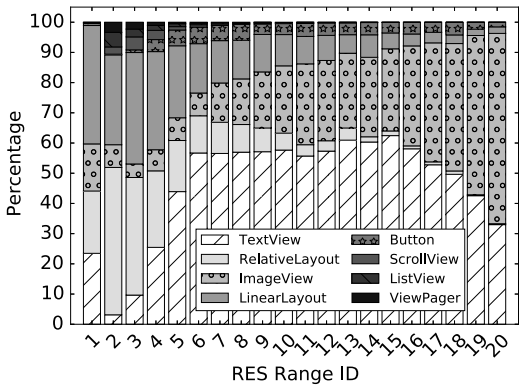


Fig. 5. Overall results of RES. Most `ImageViews` are in the right part of the figure, i.e., with high RES. Therefore, the rendering efficiency of `ImageView` is high. Similarly, the rendering efficiency of `RelativeLayout` is low.

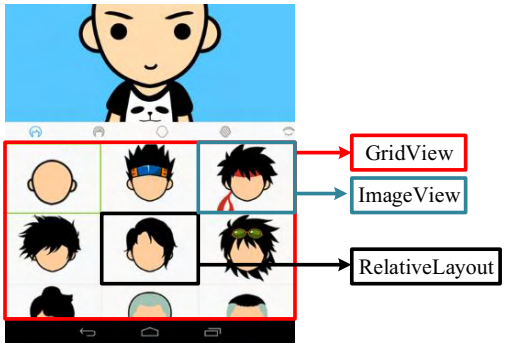


Fig. 6. Screenshot of FaceQ. There are one `GridView`, nine `ImageViews` and nine `RelativeLayouts` in the bottom part of the screenshot. The `GridView` at the background is set to be blue. However, the front `ImageViews` complete cover the background, making the draw operation of the background inefficient.

efficiencies of `RelativeLayout` and `LinearLayout` are quite low. It is reasonable since a `layout` itself usually does not include any visible pixels. However, rendering these `layouts` takes a large portion of rendering time.

B. Case Study: Repeated-draw Behaviors

Case study 1. In this case study, we use DRAW to analyze an app called FaceQ. FaceQ is a powerful avatar making app with more than 10M downloads. We use DRAW to analyze the overdraw performance of FaceQ, i.e., calculate the RES for each view object. In the RES results, the RES of a `GridView` is very small. Figure 6 shows this `GridView`, with other overlapping `ImageViews` and `RelativeLayouts`. It is obvious that the `GridView` itself is completely invisible. However, the RES of this `GridView` is very small, indicating that it consumes a relative long time to render this `GridView`[1]. Since DRAW can map each view object to the source code (Section III), we are able to find out the root cause. In file `EditorActivity.java`, the `GridView` is configured to have a blue background, by `mGridView.setBackgroundResource(R.color.gridview_background_blue)`. Therefore, the rendering inefficiency of this `GridView` has been accurately located by DRAW.

In this case study, we also modified (decompile, modify and rebuild) the app according to the UI rendering analysis results. Concretely, we modified the background setting of the `GridView` and evaluated the UI rendering performance of the app again. We run the original app and the modified app five times and report the average results. In these tests, we generated user inputs with respect to the modified part of the app. Results show that the RES of the `GridView`, the `ImageViews` and the `RelativeLayouts` are improved by 5.3%, 13.3%, and 8.3%, respectively. In particular, the number of dropped frames is decreased by 16.5% on average after the modification (from 105.4 dropped frames to 88.0 dropped frames on average).

Implication: Do not set background color for a view when it is completely covered by its children views.

[1]Note that during the trace preprocessing (Secion III), the rendering time of these `ImageViews` and `RelativeLayouts` has been removed from the rendering time of the `GridView`.

C. Case Study: Invisible-draw Behaviors

Case study 2. We use DRAW to analyze the rendering efficiency of a photo editing app, PicsPlay. Figure 7 shows 15 view objects with the smallest RES values. The first column includes the class name of the view object, and the second column includes the XML file where the view is defined. In these 15 view objects, the first 12 view objects are related to `main_submenux`. Figure 8 shows the screenshot including these views. Only one submenu is visible a time, but all the other submenus are also rendered, degrading the rendering efficiency significantly. The following several view objects are advertisement-related objects. Since ads are displayed after network fetching, it rendering efficiency degrades in case of poor network connection. From this case study, we can see that DRAW can easily pinpoint the view objects with inefficient rendering operations.

Similar to case study 1, we also modified the app and evaluate the UI rendering performance of the app. Since the source code of the app is not available, we could only modify some coarse-grained settings (e.g., XML files) of the views identified by DRAW to improve the rendering performance. Results show that the average RES improves by 10.7% and number of dropped frames is decreased by 26.9% on average after the modification (from 46.8 dropped frames to 34.2 dropped frames on average). We believe that app developers can further improve the UI rendering performance by directly modifying the source code of their apps.

Implication: Do not render all submenus when only one is visible at a time.

Case study 3. In this case study, we use DRAW to analyze the rendering performance of Pregnancy+, which is an app providing professional advices and service for those who has a pregnant family member. In this case study, the RES of many `TextViews` are very small. The reason is that many `TextViews` are out of screen, degrading the rendering efficiency. DRAW further locates the source code loading these `TextViews` as `{this.mMainView = inflater.inflate(...); initDB(); initUI();}`. From the code snippet, we can see that the whole article is loaded from a database query, which is inefficient and may cause excessive response time. In order to fix the problem, we need to change the layout

android.widget.TextView	res/layout/main_submenu2.xml
android.widget.TextView	res/layout/main_submenu3.xml
android.widget.TextView	res/layout/main_submenu3.xml
com.android.internal.widget.ActionBarView $HomeView	res/layout/main_submenu4.xml
android.widget.TextView	res/layout/main_submenu4.xml
com.android.internal.widget.ActionBarView	res/layout/main_submenu4.xml
com.google.android.gms.ads.AdView	res/layout/main_submenu5.xml
android.widget.TextView	res/layout/main_submenu5.xml
android.widget.TextView	res/layout/main_submenu5.xml
com.google.android.gms.ads.AdView	res/layout/main_submenu6.xml
com.google.android.gms.internal.v$a	res/layout/main_submenu6.xml
android.widget.TextView	res/layout/main_submenu6.xml
com.google.android.gms.ads.AdView	res/layout/activity_main.xml
com.google.android.gms.internal.v$a	res/layout/activity_main.xml
com.google.android.gms.ads.AdView	res/layout/activity_main.xml

Fig. 7. 15 view objects with the smallest RES values.

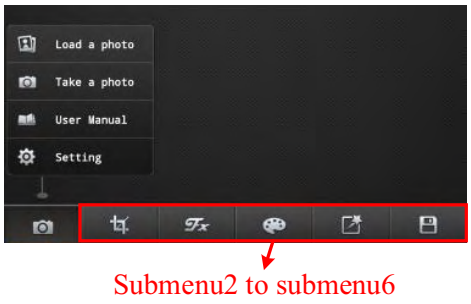


Fig. 8. Screenshot of app PicsPlay. 5 submenus are rendered but not displayed.

design and the database accessing method of the app. It is too complex to conduct such modifications without the source code. However, the app developers can easily fix these problems with the source code.

Implication: Do not render a large view when most of it is out of screen.

Implication: Be careful when performing synchronized database query in UI thread.

D. Case Study: Rendering Performance Issue Diagnosis

Case study 4. In this case study, we use DRAW to diagnose rendering performance issue of the same app used in case study 3, Pregnancy+. We run the app with DRAW enabled and obtain the RCS values of all view objects using DRAW. From the RCS results, we notice that the following view object has high RCS in traces collected from multiple devices, `com.hp.pregnancy.lite.baby.timeline.TimelineScreen$MyTimeLineGallery`. We then inspect the features with high contributions to this view object. Results indicate that the feature *draw_time_avg* contributes the most. Therefore, the long rendering time is mainly caused by long *draw* operations. Since image loading is one of the most common operation which requires a long draw operation, we further inspected the image loading code located by the source code mapping method in DRAW (Section III-B). Figure 9 shows the related code snippet. We can see that a bitmap is loaded and resized for display (`loadBitmapAndResize`). In addition, a GC (Garbage Collection) is explicitly called when loading the bitmap (`System.gc()`). These operations

```
public View getView(int index, View view, ViewGroup viewGroup) {
    ...
    this.mMeterBitmap = loadBitmapAndResize(...);
    return i;
}

public Bitmap loadBitmapAndResize(String file, float scale, float rotate) {
        ...
        BitmapFactory.decodeFile(file, options);
        ...
        Bitmap tmpBmp = BitmapFactory.decodeFile(file, options);
        Bitmap resizedBmp;
        if (desiredScale != 1.0f) {
            resizedBmp = getResizeBitmap(tmpBmp, desiredScale, rotate);
            tmpBmp.recycle();
            System.gc();
            return resizedBmp;
        } else if (rotate == 0.0f) {
            return tmpBmp;
        } else {
            resizedBmp = getResizeBitmap(tmpBmp, desiredScale, rotate);
            tmpBmp.recycle();
            System.gc();
            return resizedBmp;
        }
        ...
}
```

Fig. 9. Code snippet about rendering.

are all time consuming. Therefore, the root causes of the long response time we observed of this app are successfully located.

Similar to case study 3, the UI rendering inefficiency in this case study is too complex to be fixed without source code of the app. However, with the source code, developers can easily improve the UI rendering efficiency. For example, changing the image loading and resizing to be asynchronous could improve the UI rendering efficiency.

Implication: Synchronized image loading and frequent explicit GC could harm the responsiveness of an app.

VII. Related Work

Due to the importance of mobile application performance, many approaches have been proposed in the literature to measure and optimize it. Since network and storage I/Os may cause excessively long delays and significantly decrease user experiences, most of existing approaches focus on these two kinds of operations [20], [21], [8], [13], [17], [10], [14], [12], [16].

Many approaches have focused on delays introduced by network I/Os over the last few years [20], [21], [8], [13], [17], [10], [14]. For example, AppInsight [20] is a system that instruments mobile application binaries to automatically track execution paths and response times after user inputs, across asynchronous-call boundaries. After analyzing the tracked paths, developers can locate some critical paths and use them to optimize the performance of their apps. Since storage accessing could also cause long delays, there are many approaches focusing on measuring and optimizing storage I/O delays. For example, researchers revisit the storage system on smartphone and reveal strong correlations between the storage system design and the performance of many mobile applications [12]. Based on the findings, the authors also implement and evaluate a set of solutions to address the storage performance issues.

There are also research studies about mobile application performance from different perspectives. Since resource usage is important to application performance, ARO [19] exposes the cross-layer interaction to enable the discovery of inefficient

resource usage. In order to diagnose mobile app QoE (quality-of-experience), QoE Doctor [9] supports accurate, systematic, and repeatable measurements and analysis of mobile app QoE. PerfChecker [15] is a static code analyzer which can detect performance bug patterns. FALCON [23] exploits predictive user context to speed up application launching. Tango [11] replicates an application and executes it on both the client and the server, obtaining up to 3x speedup.

Different with the above approaches, DRAW focuses on the UI rendering performance of mobile applications. Despite the rich literature of performance research for mobile applications, only a few of them have studied the UI rendering performance. For example, in PFR [6], researchers propose a technique called parallel frame rendering which exploits the high degree of similarity between consecutive frames, to trade responsiveness for energy efficiency. The Android system provides an "overdraw" detection tool. However, it can only display the *repeated-draw* behavior of the current screen in a coarse-grained manner. Researchers also exploit the frame redundancy and propose a task level memoization scheme which improves the GPU rendering speed significantly [7]. Different with these approaches, DRAW is a UI rendering analysis tool which can provide fine-grained overdraw analysis and UI rendering performance diagnosis for developers. The above rendering-related approaches are orthogonal to DRAW.

VIII. Conclusion

In this paper, we propose DRAW, a UI rendering analysis tool for mobile applications. Given an application binary, DRAW automatically runs the applications and records detailed UI rendering data. Based on the recorded data, DRAW performs overdraw analysis and performance diagnosis, and generates two ordered lists of view objects with performance problems. Then the developer can improve the UI rendering performance of their mobile applications by optimizing/redesigning the pinpointed view objects. Comprehensive evaluations are conducted using 1,158 mobile applications. Results show that DRAW can accurately pinpoint the view objects with performance issues. Four case studies show how DRAW can help developers improve the UI rendering performance of their apps.

References

[1] *HiroMacro Auto-Touch Macro*. [Online]. Available: https://goo.gl/nINTRn

[2] *Number of apps available in leading app stores as of May 2015*. [Online]. Available: http://goo.gl/tbPJnc

[3] *Tecmark Survey about Smartphone Usage*. [Online]. Available: http://goo.gl/gWa2tA

[4] *UI Automator*. [Online]. Available: https://goo.gl/oiBHvk

[5] *Worldwide Smartphone Shipments Top One Billion Units*. [Online]. Available: http://goo.gl/aryvyO

[6] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Parallel frame rendering: Trading responsiveness for energy on a mobile GPU," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[7] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[8] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar, "KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[9] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, "QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis," in *Proceedings of Conference on Internet Measurement Conference (IMC)*, 2014.

[10] V. Gabale and D. Krishnaswamy, "MobInsight: On Improving The Performance of Mobile Apps in Cellular Networks," in *Proceedings of International Conference on World Wide Web (WWW)*, 2015.

[11] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," in *Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[12] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting Storage for Smartphones," *Trans. Storage*, vol. 8, no. 4, pp. 14:1–14:25, 2012.

[13] N. Larson, D. Baltrunas, A. Kvalbein, A. Dhamdhere, K. Claffy, and A. Elmokashfi, "Investigating Excessive Delays in Mobile Broadband Networks," in *Proceedings of ACM SIGCOMM 2015 Workshop on All Things Cellular: Operations, Applications and Challenges*, 2015.

[14] W. Li, R. K. P. Mok, D. Wu, and R. K. C. Chang, "On the Accuracy of Smartphone-based Mobile Network Measurement," in *Proceedings of International Conference on Computer Communications (INFOCOM)*, 2015.

[15] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2013.

[16] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang, "Reducing Smartphone Application Delay through Read/Write Isolation," in *Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[17] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao, "Mobilyzer: An Open Platform for Controllable Mobile Network Measurements," in *Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[18] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proceedings of ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2013.

[19] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Profiling Resource Usage for Mobile Applications: a Cross-layer Approach," in *Proceedings of International Conference on Mobile Systems (MobiSys)*, 2011.

[20] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile App Performance Monitoring in the Wild," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementatio (OSDI)*, 2012.

[21] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling User-perceived Delays in Server-based Mobile Applications," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[22] Y. Ren, C. Wang, J. Yang, and Y. Chen, "Fine-grained sleep monitoring: Hearing your breathing with smartphones," in *Proceedings of International Conference on Computer Communications (INFOCOM)*.

[23] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast App Launching for Mobile Devices Using Predictive User Context," in *Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[24] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. A. Dinda, "Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance," in *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.