electronics

MDPI

Article

Scrolling-Aware Rendering to Reduce Frame Rates on Smartphones

Woo-Hyun Ahn [1], Chang-Kyung Hong [1], Kyung-Min Han [1], Seong-Hun Choi [1], Jae-Won Oh [2] and Seung-Ho Lim [3,*]

[1] School of Software, Kwangwoon University, Seoul 01897, Korea; whahn@kw.ac.kr (W.-H.A.); dominicong@naver.com (C.-K.H.); qhwhehwh@naver.com (K.-M.H.); skyworking@naver.com (S.-H.C.)
[2] School of Computer Science and Information Engineering, The Catholic University of Korea, Bucheon-si 1466, Korea; jwoh@catholic.ac.kr
[3] Division of Computer Engineering, Hankuk University of Foreign Studies, Yongin-si 17035, Korea
* Correspondence: slim@hufs.ac.kr; Tel.: +82-31-330-4704

Abstract: One of the major sources of power drain in smartphones is a frame rendering and display process called graphics pipeline, in which power consumption depends largely on frame rendering operations per second (fps), known as the frame rate, and the quantity of UI content to be rendered. We discovered a major problem causing power consumption upon a scrolling operation: The Android graphics pipeline renders all or a large portion of the content displayed most recently at a frame rate of nearly 60 fps. This paper proposes a scrolling-aware rendering (SCAR) scheme to reduce the frame rate caused by a scrolling. When rendering a frame for UI content to be displayed, SCAR pre-renders UI content that is likely to be displayed soon in any subsequent scrolling operation. This frame is extended to place the pre-rendered UI content contiguously with the UI content to be displayed. Upon a subsequent scrolling, SCAR repositions the extended frame on screen by a scrolling distance instead of rendering a new frame. Our experiments on a smartphone show that SCAR reduced frame rates to below one fps in scrolling, thus saving power by up to 30%.

Keywords: android operating system; android graphics pipeline; smartphone; power consumption; frame rate; frame rendering; scrolling

check for updates

Citation: Ahn, W.-H.; Hong, C.-K.; Han, K.-M.; Choi, S.-H.; Oh, J.-W.; Lim, S.-H. Scrolling-Aware Rendering to Reduce Frame Rates on Smartphones. *Electronics* **2021**, *10*, 2177. https://doi.org/10.3390/electronics10172177

Academic Editor: Tony Givargis

Received: 4 July 2021
Accepted: 2 September 2021
Published: 6 September 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Recently, smartphones have become equipped with high-performance CPUs, GPUs, and display panels, but battery capacity has not yet met the increasing power demands of the components. As display panels support higher screen resolutions and color gamuts to improve the user experience, they become major power consumers [1,2]. Previously, backlight scaling [3], dynamic voltage scaling [4,5] and low-power color transformation [6] were introduced to reduce the power consumption of the display panel, but these deteriorated the user experience. Moreover, a frame rendering and display process, called a *graphics pipeline*, renders frames for UI content (content for short) by up to 60 times per second to smooth the user experience. The frame rate indicates the frequency of frame rendering operations per second (fps). However, the higher the frame rate, the larger the CPU and GPU usages of the graphics pipeline, which consequently increases power consumption [7,8].

Power consumption in the graphics pipeline depends on the quantity of the content to be rendered with the changes in content [9], together with the frame rate. The most common use pattern causing changes in content is scrolling, which accounts for a high percentage of use patterns in interactive apps [8]: the ratio of the scrolling time to the overall time of app usage is higher than 30% and up to 60%. We found two important problems caused by scrolling. First, the Android graphics pipeline [10] unnecessarily changes all or a large portion of the content displayed most recently even if there are only

changes in their on-screen location. Second, frames are frequently rendered at a frame rate of nearly 60 fps to display the unnecessarily changed content smoothly.

There have been studies on saving power consumption from the perspective of the content changes in the Android graphics pipeline. Lee [7] displays a previously rendered frame instead of rendering a new frame if there was a high similarity among frames rendered in the recent past. However, that study cannot reduce power consumption in use patterns that cause a change in content, such as scrolling. Kim [11] dynamically lowers a refresh rate if there is a high similarity among consecutive frames displayed in the recent past, but it can degrade the user experience if there is a sudden change in content. Seo [9] reduces a region in which content is changed, but deteriorates the user experience because content outside the region is not rendered. Yu [8] adjusts frame rates upon scrolling by predicting the minimum acceptable frame rates to avoid degrading the user experience according to scrolling speed. However, this method can degrade the user experience if the model is not accurate.

This paper proposes a scrolling-aware rendering (SCAR) scheme to reduce frame rates caused by scrolling without compromising the user experience. SCAR is inspired from an adaptive tile refresh [12] that was introduced in the early 1990s to reduce redrawing operations upon scrolling in video games in desktop PCs. When rendering a frame for content to be displayed on screen, SCAR pre-renders content that is likely to be displayed soon in a subsequent scrolling action. This frame is then extended to place the pre-rendered content contiguously with the content to be displayed. In any subsequent scrolling, SCAR repositions the extended frame on the screen by a scrolling distance, without needing to render a new frame. This reposition can significantly reduce frame rates resulting from scrolling. Moreover, our scheme has an important feature to be applicable to various display panels regardless of the type of display technology because it attempts to lessen frame rates to reduce CPU and GPU overheads caused by frame rendering operations.

Our scheme has been implemented in the Android 9 operating system whose source code is provided by the Android Open Source Project (AOSP) [13]. An evaluation on a Google Pixel 3XL smartphone shows that SCAR reduced the average frame rate to below one fps by reusing frames in scrolling. This reduction lessens CPU and GPU overheads by up to about 65% and 100%, respectively, compared to the Android graphics pipeline, thus reducing power by up to 30%.

The contributions of this paper are as follows. First, we discovered the problem that scrolling causes rendering of most or all of the content displayed most recently at a frame rate of nearly 60 fps. Second, our study is the first attempt to pre-render content in the Android graphics pipeline to improve power consumption during scrolling. Third, we detail the overall execution flow, including scroll-event handling, frame rendering, frame memory management, and frame display processes, to help researchers understand the Android graphics pipeline. Finally, our study is the first to perform a quantitative analysis of the CPU and GPU overheads caused by scrolling and of the relationships between power consumption and these overheads.

The remainder of this paper is organized as follows. Section 2 introduces the Android graphics pipeline as background. Section 3 introduces related works and motivation, respectively. Section 4 presents the design and implementation details of SCAR. In Section 5, we evaluate the performance and overhead of SCAR. Section 6 discusses the implications for research and practice, real-world applicability, limitations, and future work. Finally, we conclude this paper in Section 7.

2. Background

The Android graphics pipeline [10] conducts a sequence of rendering, coordinating, and composing operations. An app renders a frame, a system service called SurfaceFlinger coordinates the on-screen layout of the frame, and a hardware composer (HWC) in a display driver composes the frames of the app and system UIs into an image for display.

2.1. Frame Rendering

As shown in Figure 1a, a screen is generally composed of three windows, the app, status bar, and navigation bar, each of which is a rectangular region consisting of UI elements called views (e.g., button, text). A window has a view tree, a surface, and a buffer queue to manage views and render frames for them. A view tree manages views hierarchically, each of which has its properties, such as height, width, and an object called *render node* with drawing commands. A surface serves as a drawing canvas on which views are drawn, containing a buffer queue, a pool of three graphic buffers in which to store rendered frames. This triple buffering is to avoid the screen-tearing problem. The following components in an app cooperate to perform the rendering process: a UI thread manages views in a view tree, a render thread renders a frame for views to be displayed, and a BufferQueue producer (BQProducer) manages a buffer queue.

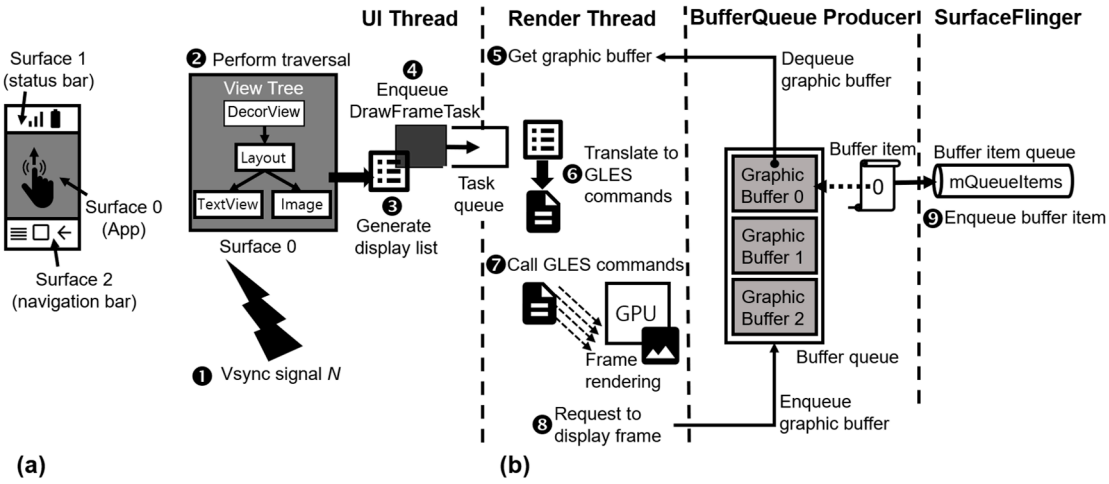

Figure 1. Frame rendering process of the Android graphics pipeline: (**a**) surfaces; (**b**) frame rendering flow.

Figure 1b illustrates the frame rendering process. When user input changes views, the UI thread invalidates the views to indicate that they need to be redrawn. The rendering of these views starts with a V-Sync signal (step 1) that HWC generates every 16.7 ms, 60 times per second. The UI thread first traverses a view tree from a root view called *DecorView* to find render nodes for views to be drawn (step 2). A render node has a display list consisting of a sequence of drawing commands that specify how to render a view. A drawing command is an intermediate operation mapped onto a complex set of commands in OpenGL ES (GLES) [14], a platform for rendering 2D and 3D graphics in embedded systems. The UI thread places display lists of the found render nodes into a rendering request called *DrawFrameTask* (step 3) that tells the render thread how to render a frame. This request is transferred to a task queue of the render thread (step 4).

The render thread starts rendering a frame when it receives a rendering request from the task queue. It dequeues an available graphic buffer (grbuffer) into which to store a frame, with the help of BQProducer (step 5). Drawing commands in the request are translated into the corresponding GLES commands (step 6), which are executed to make the GPU render a frame into the grbuffer (step 7). Upon the request to display the frame, BQProducer puts the grbuffer into a buffer queue (step 8) and composes a message, called *buffer item*, of the display information including the grbuffer number (e.g., 0). This message is sent as a request to display the frame to SurfaceFlinger via a buffer item queue (step 9). Note that the render thread cannot know the details of the buffer queue, such as which grbuffer is dequeued, enqueued, or stores a rendered frame because, during frame rendering, the enqueue/dequeue operations are invoked by GLES commands in the GLES library, which is an external non-open-source library. This limitation is an important

challenge in implementing our scheme of modifying graphics components that are linked with each other via the buffer queue.

BQProducer and SurfaceFlinger share a grbuffer in a buffer queue through its grbuffer number. BQProducer manages the information about grbuffers in an array called *mSlots*, each entry containing the handle and state of a grbuffer indexed by a grbuffer number. A grbuffer handle refers to an object containing a grbuffer height, width, and address that points to a native window buffer allocated by the graphics memory allocator. BQProducer dequeues a grbuffer with a "free" state, in the order of the grbuffer numbers, 0, 1, and 2, and changes the state to "dequeued," meaning that the grbuffer is being used for rendering a frame. After this rendering, the grbuffer is enqueued with the "queued" state, meaning that it is filled with a frame. SurfaceFlinger changes the state to "acquired" to prevent other components from accessing the grbuffer in displaying the frame. After the frame is displayed, the grbuffer is released with the "free state", meaning that it is available for a frame rendering.

2.2. Frame Coordination and Composition

SurfaceFlinger coordinates the frames of three windows for a user app, navigation bar, and status bar according to their sizes and positions on the screen and requests HWC to combine these frames into one image. In Figure 2a, SurfaceFlinger manages the surfaces of several windows as layers, each of which has a depth level in the on-screen Z-order: layer 0 and layer 2 have the lowest and highest depth levels, respectively. In Figure 2b, the frame coordination starts with a V-Sync signal (step 1) generated after the completion of the frame rendering process in Figure 1b. Using an array managing layers, SurfaceFlinger searches a layer that has been asked to change its layout, such as its size or position, before the V-Sync signal (step 2). If the layer is found, its layout changes are reflected into the member variable *mDrawingState* of the layer that manages the layout information (step 3). This variable is used hereafter to determine the size and position of the layer when HWC composes the layer.

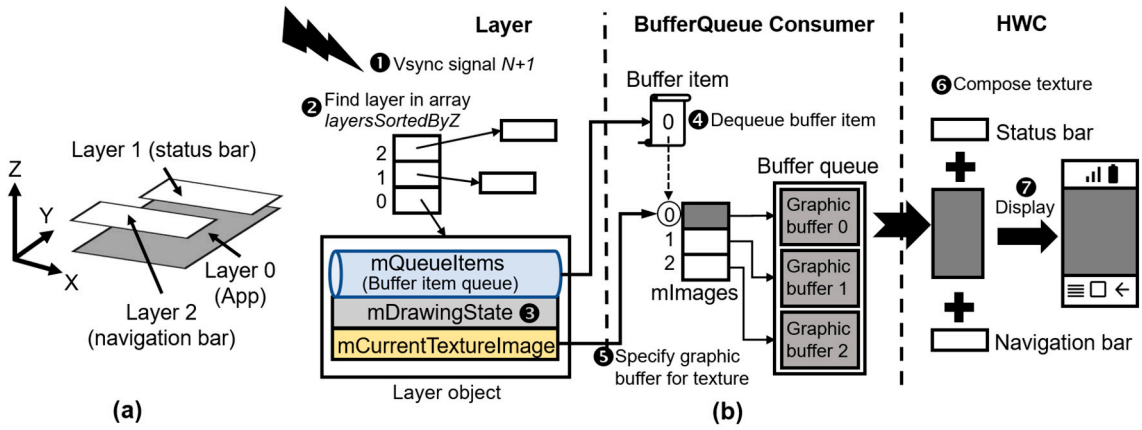

Figure 2. Frame display process of SurfaceFlinger: (**a**) layers; (**b**) frame display flow.

It is necessary to create a GLES texture through a frame (or image data) in a grbuffer to display the frame rendered with GLES commands. In computer graphics, a texture contains a texture image or picture that can be mapped or painted onto the exterior of a geometric shape, such as a polygon or flat canvas. The creation of a texture requires designating which image data is to be used as a texture image. This designation extracts colors from image data and copies them into dedicated GPU memory to help the GPU accelerate the texture mapping. In contrast, the GPU and CPU share memory, due to the lack of memory in embedded systems such as smartphones, in which GLES supports a resource called *EGLImage* [15,16] to designate image data or a grbuffer to a texture without copying any memory. In the Android graphics pipeline, SurfaceFlinger creates an EGLImage through a

grbuffer and binds the EGLImage to a texture. This binding enables the GPU to access the frame in the grbuffer directly.

A BufferQueue consumer (BQConsumer), a component in SurfaceFlinger, acquires a grbuffer from a buffer queue and designates the grbuffer as a texture with the following procedure. It first receives a buffer item delivered from BQProducer to obtain grbuffer number n (e.g., 0) (step 4), which is used as an index into an array named *mImages* in BQConsumer. SurfaceFlinger implements the array in its local memory so that BQConsumer can access a grbuffer by n. An entry in the array contains the handle and state of a grbuffer, and an EGLImage that is created with the grbuffer. BQConsumer designates a grbuffer indexed by n as a texture (step 5) by assigning mImages [n] to its variable *mCurrentTextureImage*, which specifies a grbuffer with which to make and coordinate a texture as follows. The first step is to create an EGLImage through the grbuffer and bind the EGLImage to a texture, and the second is to coordinate the texture layout through the layout information in the mDrawingState. HWC composes the texture, the navigation bar, and status bar into one image according to their Z-order levels (step 6). Finally, the image is displayed on the screen (step 7).

3. Related Work and Motivation

3.1. Related Work

In the early 1990s, Carmack [12] introduced an adaptive tile refresh to minimize drawing operations for scrolling video games in PCs. The method makes a buffer wider and taller than a screen in a video memory of a graphic card and draws content outside the edges of the screen into the buffer, together with on-screen content. In any subsequent scrolling, the content in the buffer is repositioned on the screen. However, the method was not applicable to general apps other than video game apps allowing low-resolution images because of deteriorating the user experience.

There have been many studies on reducing power consumption in mobile devices. In the field of display panels, Dong [6] proposed color transformation to convert the colors of content into those with low power consumption. Tan [17] inferred an on-screen region that a user focuses on and darkens content outside the region. Dynamic voltage scaling [4,5] adjusted the voltage supplied to a display depending on content features. However, the studies degrade the user experience significantly when there are frequent changes in content.

Some studies have attempted to eliminate unnecessary frame rendering for unchanged content. Lee [7] found that frames are unnecessarily rendered even if there is no change of content. His method compares the display lists needed to render the current frame with those used to render the previous frame. If they are the same, the previous frame is displayed instead of rendering a new frame. Hwang [18] solved the problem of frames being rendered at 60 fps in mobile game apps, even if there is no change in content, by displaying a previous frame if there is a high similarity among the frames rendered in the recent past. These studies, however, have no effect on reducing frame rates in use patterns such as when scrolling causing changes in content.

Some studies have saved power consumption using the features of human perception. Egilmez [19] proposed a model that predicts a user's satisfaction with each frame rate and minimized the frame rate depending on that satisfaction. However, His method has a negative impact on the user experience if there is an error in predicting the satisfaction. He [20] proposed a model to calculate the minimum resolution that does not degrade the user experience according to the distance to the user. With this model, the resolution is lowered in proportion to the distance to reduce GPU overhead caused by frame rendering at high resolution. However, a smartphone needs to be equipped with an ultrasonic sensor to calculate the distance to a user.

There have been efforts to control the refresh rate, i.e., the speed of updating a frame to the screen. Nixon [21] measured power consumption according to the refresh rate and introduced a scheme to adjust the refresh rate dynamically. However, the scheme was not

verified with an implementation. Kim [11] observed that frames are displayed at a 60-Hz refresh rate even if there is no change among the frames. His method reduces the refresh rate if there is a similarity among the frames displayed in the recent past, but can degrade the user experience if there is a sudden change in content.

Yu [8] attempted to reduce the frame rate caused by scrolling. His method makes a model of predicting a minimum frame rate that satisfies the user experience according to scrolling speed. With this model, frames are rendered at a minimal frame rate for each scrolling speed. However, a remote server is needed to rebuild a highly accurate model via a user's feedback. There can be a drop in the user experience until the model becomes highly accurate. Unlike our study, this study fails to find which behaviors incur CPU overhead in frame rendering because of not handling power consumption problems in the graphics pipeline perspective. Moreover, there is no analysis on the impact of GPU overhead on power consumption.

There were studies in adjusting CPU/GPU frequencies on game apps. One study [22] dynamically lowered CPU/GPU frequencies as the frame rate decreased. Pathania [23] estimated power consumption and frame rates according to CPU/GPU frequency and predicted the optimal frequencies for each frame rate. Chen [24] monitored the execution of GPU instructions in real time to lower GPU frequency under a low GPU workload. These studies, however, help to reduce the power consumption only in mobile game apps.

Finally, the problem of power consumption upon scrolling has not been introduced in the popular mobile OS, the Apple iOS whose source code cannot be analyzed due to the non-open source policy. This policy makes it difficult to examine an impact of scrolling on power consumption and understand how to handle the scrolling problem. The iOS provides APIs [25] of asynchronously prefetching content to be displayed soon in scrolling over the network. The APIs are to support displaying the content at the frame rate of 60 fps, but not improving power consumption.

3.2. Motivation

We activated the "Show GPU view updates" menu in the Android Settings app to examine how many views are drawn in a scrolling operation. This activation flashes views that the GPU renders on the screen. The investigation in commercial apps revealed that a large portion of the views on the screen continuously flickers even when the scrolling distance is small. We also measured the number of views to be drawn to answer the following question: upon scrolling, does the Android graphics pipeline really redraw many of the views displayed most recently? This measurement is made when the UI thread traverses a view tree, finds render nodes for views to be drawn, and obtains display lists from the render nodes. Table 1 shows the number of views in a view tree and number of the views to be redrawn in the view tree. We discovered the following two facts: the first is that a view tree in an app consists of many views, reaching at least 110 views and up to 959, and the second is that scrolling forces at least 51% of all the views to be redrawn even if there is no change except for their positions. This redrawing can cause unnecessary CPU and GPU overheads, thus increasing power consumption.

Table 1. Number of views in a window and number of views to be redrawn in a scrolling operation.

App Name	Number of Views	Number of Views to Be Redrawn
Wish	959	959
Twitter	417	369
Hotels.com	391	266
News Republic	321	205
Twitch	217	115
Facebook	155	84
Tumblr	219	115
Resplash	110	56

In Figure 3, we show the CPU and GPU utilization using the Snapdragon profiler tool [26] when scrolling through a list of hotels in the Hotels.com app on a Google Pixel 3XL smartphone. When scrolling starts, the two utilizations suddenly increase. Until the scrolling operation ends, the GPU utilization remains as high as approximately 12%, because the GPU renders hundreds of views. The CPU utilization goes up and down at approximately 5% and sometimes reaches as high as 8%. This is because the render thread traverses a view tree to obtain display lists for views to be redrawn and translates drawing commands in the display lists to GLES commands.

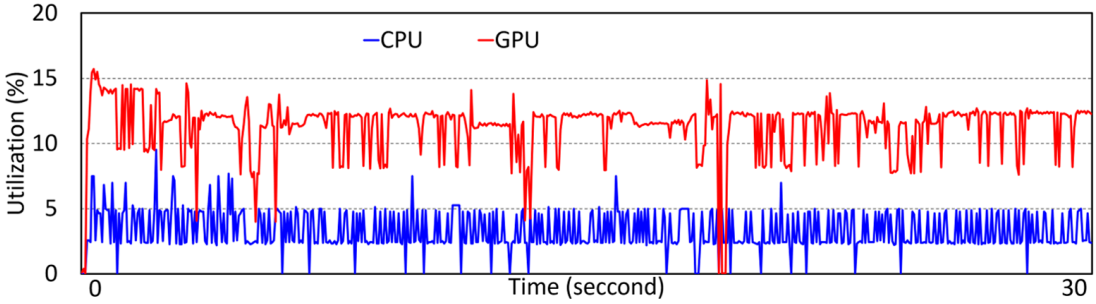

Figure 3. Variation in CPU and GPU utilization during scrolling.

Our motivation derived from a question: is it necessary to render previously rendered views again even if they only move in the Y-order direction by a scrolling distance? If we could reuse the rendered frame upon scrolling, it would be possible to reduce power consumption by eliminating unnecessary frame rendering. In a representative scenario, a user scrolls on a shopping app such as the Wish app to choose his shopping item. It is assumed to pre-render images of 10 items to be displayed soon in a subsequent scrolling when rendering images of 10 items to be displayed currently. A frame places the pre-rendered images contiguously with the images to be displayed. There can be a decrease in CPU and GPU overheads if the frame is reused without a frame rendering to display the pre-rendered images upon a subsequent scrolling. Thus, our motivation is to minimize frame rates by reusing the frame displayed most recently.

4. Scrolling-Aware Rendering

4.1. Overview

We propose the SCAR scheme to reduce the frame rate caused by scrolling on a scrollable window including a scrollable view such as the RecyclerView and ListView. In Figure 4, SCAR pre-renders content (pink regions) that is likely to be displayed soon in a subsequent scrolling when rendering a frame for the current content to be displayed. The main feature is to extend the frame size to N times the frame size (H) that is configured in the Android graphics pipeline. We term this extended frame an *exFrame*. This extension allows the frame to place the region of pre-rendered content (defined as pre-rendering region) contiguously with that of the on-screen content. N is an extending factor to determine the exFrame size, and its range is $1 < N \leq 2.5$, because N cannot exceed 2.5 due to system memory limitations. The pre-rendering is applicable to all the types of views (e.g., text, image, button) in a scrollable window. After rendering an exFrame, SCAR requests SurfaceFlinger to display only the content to be seen, excluding that of the pre-rendering region. Upon a subsequent scrolling, SCAR decides to reuse the exFrame instead of rendering a new frame and requests SurfaceFlinger to reposition the exFrame by a certain scrolling distance. Such a reuse avoids rendering new frames, thus reducing GPU and CPU overheads, which are the major causes of power consumption.

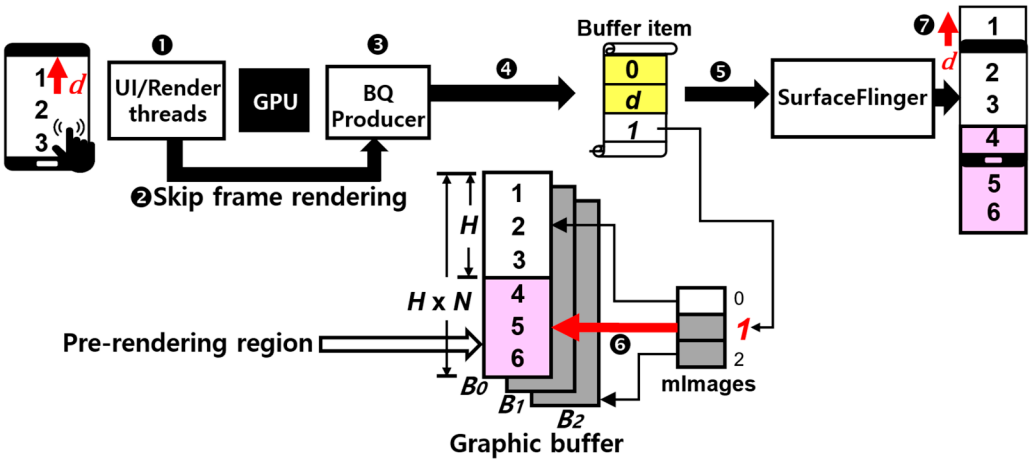

Figure 4. Overview of SCAR.

Figure 4 shows how SCAR reuses an exFrame when a user scrolls by distance d in pixels. Suppose that an exFrame has been rendered into grbuffer B_0 prior to the scrolling. We define a grbuffer storing an exFrame as an exFrame grbuffer. The UI thread obtains display lists to transfer them to the render thread, which dequeues grbuffer B_1 (step 1) as in the Android graphics pipeline. SCAR skips translating drawing commands in the display lists to GLES commands and calling these translated commands (step 2) because it reuses the exFrame from B_0. This skip prevents the GPU from rendering a valid frame into B_1, which is nevertheless put into a buffer queue (step 3). We define a grbuffer (e.g., B_1) without a valid frame (or rendered content) as a junk grbuffer. BQProducer sends a buffer item with the display information such as the enqueued grbuffer number (i.e., 1) to SurfaceFlinger (step 4). SCAR differs from the Android graphic pipeline in that d and exFrame grbuffer number 0 (yellow in Figure 4) are additionally put into the message, so that SurfaceFlinger can reposition a frame of B_0 by d.

SCAR dequeues/enqueues a junk grbuffer from/to a buffer queue even if there is an exFrame grbuffer in a buffer queue. The render thread cannot know which grbuffer stores an exFrame to be reused in scrolling because the GLES library prevents the render thread from directly accessing the buffer queue. Due this limitation, we had difficulty in finding the exFrame grbuffer at step 1 where the render thread gets a grbuffer, but succeeded in detecting the exFrame grbuffer at step 3 where BQProducer puts a grbuffer into a buffer queue. To trigger the detection of an exFrame grbuffer, SCAR allows a junk grbuffer without rendered content to be put into a buffer queue in reusing the exFrame upon scrolling.

The following actions differ from the Android graphics pipeline when SurfaceFlinger acquires a buffer item with scrolling distance d, exFrame grbuffer number 0, and junk grbuffer number 1 (step 5). The first is to replace the handle of B_1 in mImages [1] with that of B_0 to designate an exFrame in B_0 as a texture instead of an invalid frame in B_1 (step 6). This action allows B_0 to be reused in a scrolling operation. The second is to change the layout information of a texture so that HWC can reposition the texture by d in the Y-order on the screen in composing the texture (step 7).

4.2. Scrollability and Scrolling Distance

There are two important parameters to control an exFrame rendering: scrollability and scrolling distance. If a window has a scrollable view called *scrollable container*, the view is considered as scrollable: examples are the RecyclerView, ScrollView, and ListView views. During the creation of a window, SCAR registers a callback of checking the scrollability into a listener of the view tree observer that is used to detect the change of a view tree structure. When the listener detects that all the views to be drawn are added to a view

tree, it invokes the callback of checking whether each view in the view tree is a scrollable container. If there is at least one scrollable container, the window is considered scrollable.

SCAR extends a frame when a system component called *window manager* initializes the layout of a window. The component determines the position and size of a window through the sizes of the parent window, screen, and visible areas. If the window is scrollable, SCAR increases the area sizes by the extending factor N times in the layout initialization to extend both the frame and grbuffer by N times. This extension of a grbuffer increases the memory usage of a buffer queue consisting of three grbuffers, each 17 MB in size by default. If not scrollable, it is decided not to extend the frame and grbuffer. This decision is to prevent unnecessary increase in the memory usage of the buffer queue.

When a user touches a screen, the touch device driver sends a touch event to the window manager, which delivers the touch event to a foreground app. The app executes a touch-event dispatching operation, which traverses a view tree to search a view at the touch position and dispatches a motion event with the touch information to the view. SCAR calculates a scrolling distance in this dispatching operation: upon successive down and up touches, the scrolling distance is the difference in pixels between the two Y-order positions in the motion events for the down and up touches. We define an up-scrolling, a common use pattern in smartphones, as the positive direction.

4.3. Deciding Whether to Reuse exFrame

Figure 5 shows the policy of deciding whether to reuse an exFrame according to scrolling distance. In the left of Figure 5a, suppose that an exFrame is initially displayed on the screen. Upon an up-scrolling operation by distance d_1, the exFrame is moved up in the Y-order direction. A total (or accumulated) scrolling distance is defined as the sum of scrolling distances since the rendering of an exFrame, representing a starting position where content to be displayed at the top of the screen is located within the exFrame after the scrolling. The sum of the starting position and the screen height H indicates an ending position where content to be displayed at the bottom of the screen is located within the exFrame. SCAR decides to display content between the starting and ending positions, d_1 and $d_1 + H$, in the exFrame. Upon an up-scrolling operation by distance d_2 in Figure 5b, it is decided to display content between the two positions, $d_1 + d_2$ and $d_1 + d_2 + H$, in the exFrame.

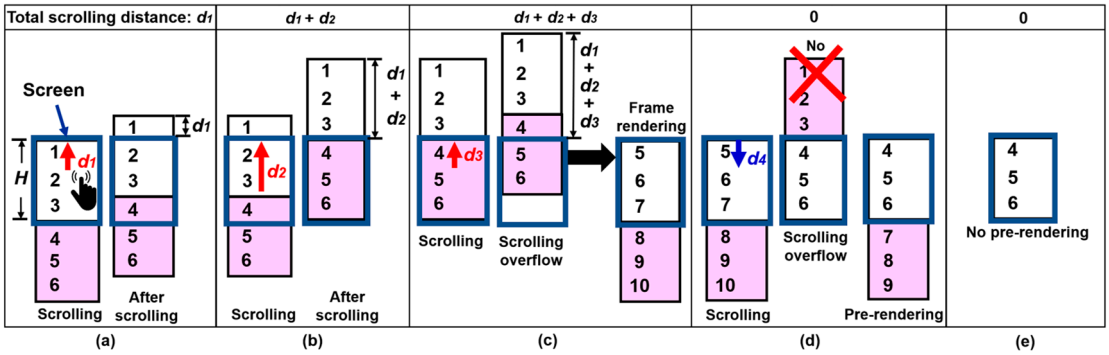

Figure 5. Policy of deciding whether to render or reuse an exFrame: (**a**) up-scrolling by d_1; (**b**) up-scrolling by d_2; (**c**) up-scrolling overflow; (**d**) down-scrolling overflow; (**e**) frame rendering in down-scrolling overflow.

Figure 5c illustrates an up-scrolling overflow, where the sum of the total scrolling distance $d_1 + d_2 + d_3$ and H goes over the ending boundary of an exFrame upon a scrolling by distance $d3$. The up-scrolling overflw means that it has been attempted to display content not contained in the exFrame, as presented in the center of Figure 5c. In this situation, SCAR renders a new exFrame consisting of content (i.e., 5 to 7) to be displayed by the scrolling and content (i.e., 8 to 10) to be pre-rendered for a subsequent scrolling,

as shown in the right of Figure 5c. Finally, the total scrolling distance is reset when an exFrame is rendered.

Figure 5d shows a down-scrolling overflow, where an exFrame does not contain content (i.e., 4) to be displayed at the top of the screen when scrolling down the screen of Figure 5c. As shown in the center of Figure 5d, it would be expected to pre-render content (i.e., 1 to 3) in the upper region of content (i.e., 4) to be newly displayed, due to the scrolling direction. As presented in the right of Figure 5d, however, SCAR cannot help pre-rendering content (i.e., 7 to 9) in the lower region of content (i.e., 6) to be newly displayed at the bottom of the screen, due to the policy of the Android graphics pipeline. This policy composes a view tree consisting of a view to be displayed at the top of the screen and other views meeting the following conditions: (1) views to be displayed below the top view in the Y-order direction, (2) many views to be contained in an exFrame. In this situation, the pre-rendered content can be reused in a subsequent up-scrolling, but not be reused in a subsequent down-scrolling, which a user is much likely to perform during continuous scrolling actions. Hence, pre-rendering the content not reusable in a down-scrolling can cause unnecessary GPU and CPU overheads. To avoid this problem, we do not modify the policy of composing a view tree, due to the complexity of the Android graphics pipeline, but render only views to be seen on screen without the pre-rendering, as shown in Figure 5e.

Algorithm 1 describes our algorithm for deciding whether to render or reuse an exFrame in the method draw() of class CanvasContext, which the render thread calls to execute the operations from step 5 to 8 in Figure 1b. This algorithm dequeues an available grbuffer (line 2) and determines a rectangular rendering region on which to update or render content (line 3). This rendering region indicates the screen region on a non-scrollable window, while a scrollable window has an extended rendering region whose width and height are the screen width and the height of N times the screen height, respectively. We define the following two parameters to decide whether to render an exFrame, together with scrolling distance and scrollability that the UI thread has passed to the method draw(). The first is a total scrolling distance, which is delivered to SurfaceFlinger to reposition an exFrame on the screen by a scrolling distance, and the second is a rendering indicator indicating whether to render an exFrame.

If a scrolling distance is negative on a scrollable window, an extended rendering region is downsized to the screen region to avoid pre-rendering views that are not reused in a subsequent down-scrolling. This downsizing is achieved by dividing the height of the extended rendering region by N (line 6). The rendering indicator is set to specify the need of rendering a frame containing only content to be displayed on the screen (line 7). If a scrolling distance is positive, the total scrolling distance summed with the scrolling distance is added to the screen height to calculate the ending position (line 11), which indicates where content to be displayed at the bottom of the screen is located within an exFrame. If the ending position does not exceed the ending boundary of the exFrame, it is decided to reuse the exFrame containing the content to be displayed (line 12); otherwise, an up-scrolling overflow occurs because the exFrame does not have some lower part of the content to be displayed. In such a case, it is decided to render a new exFrame by setting the rendering indicator (line 14). The total scrolling distance is reset regardless of the scrolling direction due to the rendering of a new exFrame (lines 8 and 15).

If either a window is not scrollable or it is decided to render an exFrame, content is rendered onto a rendering region (line 20). The actual frame rendering is performed by the Skia graphics engine [27] that serves as an intermediary between the render thread and GLES library to supports 2D graphics. This Skia rendering, however, incurs large CPU and GPU overheads of translating display lists to GLES commands and executing the GLES commands to make GPU render a frame into a grbuffer. Upon an up-scrolling, our algorithm avoids the Skia rendering by reusing an exFrame, thus decreasing the two overheads causing power consumption. Finally, a grbuffer swapping operation (line 22) enables BQProducer to enqueue either an exFrame grbuffer if an exFrame has been rendered

or a junk grbuffer if it has been decided to be reused. SurfaceFlinger is requested to swap a frame in the exFrame grbuffer with one being currently displayed. Note that the two operations in lines 2 and 22 make the Skia graphics engine execute GLES APIs, which request BQProducer to dequeue/enqueue a grbuffer from/to a buffer queue.

Algorithm 1. Algorithm of deciding whether to render or reuse an exFrame

Input : Scrolling distance *distance*, scrollability of window *scrollable*, screen height *screenHeight*, extending factor N
Output: Rendering indicator *isRendered*
1 // assume that *totalDistance* is a global variable indicating total scrolling distance
2 get Frame object *frame* with grbuffer *grBuf*
3 determine rendering region *rendRegion* $\subseteq$ *grBuf*
4 **if** *scrollable* = *true* **then**
5 **if** *distance* < 0 **then**
6 height of *rendRegion* $\leftarrow$ height of *rendRegion*/N
7 *isRendered* $\leftarrow$ true
8 *totalDistance* $\leftarrow$ 0
9 **else**
10 *totalDistance* $\leftarrow$ *totalDistance* + *distance*
11 **if** *totalDistance* + *screenHeight* $\leq N \times$ *screenHeight* **then**
12 *isRendered* $\leftarrow$ false
13 **else**
14 *isRendered* $\leftarrow$ true
15 *totalDistance* $\leftarrow$ 0
16 **end**
17 **end**
18 **end**
19 **if** *scrollable* = *false* ***or*** *isRendered* = **true then**
20 render content onto *rendRegion* for *frame* with the help of the Skia engine
21 **end**
22 swap *frame* with that being displayed on screen

4.4. Requesting the Display of exFrame and Designating exFrame as Texture

There is a challenge when a grbuffer swapping operation requests SurfaceFlinger to display an exFrame to be reused upon an up-scrolling. The Android graphics pipeline adopts a triple buffering with three grbuffers, among which BQProducer dequeues and enqueues a different grbuffer for every rendering in the order of grbuffer numbers 0, 1, and 2. However, the render thread executing Algorithm 1 cannot know which grbuffer among the three grbuffers contains an exFrame due to the interference of the GLES library to BQProducer. Moreover, BQProducer cannot know whether a grbuffer put into a buffer queue is either a junk or an exFrame one because it expects that the grbuffer always has a valid frame. These limitations make it difficult for BQProducer to identify exFrame grbuffer number n_{ex} that SurfaceFlinger needs to make a texture to be displayed on the screen.

SCAR makes the decision of n_{ex} in an enqueue operation, which BQProducer conducts to put a grbuffer into a buffer queue and send a buffer item with the information about the grbuffer to SurfaceFlinger. For the decision of n_{ex}, we provide the enqueue operation with the rendering indicator obtained in Algorithm 1. When BQProducer is requested to enqueue either a junk grbuffer or an exFrame grbuffer, it executes the enqueue operation with its grbuffer number n. In this execution, SCAR checks the rendering indicator to determine whether an exFrame has been rendered. If rendered, n indicates n_{ex} for a rendered exFrame, and its value is backed up to keep track of the exFrame to be reused in a subsequent scrolling. If not rendered, n indicates a junk grbuffer number because Algorithm 1 has decided to reuse an exFrame. In such a case, the backed-up value is used to identify n_{ex} for the exFrame to be reused. Finally, when BQProducer sends a buffer item with n to SurfaceFlinger, SCAR piggybacks n_{ex} and a total scrolling distance onto the buffer item.

In SurfaceFlinger, BQConsumer designates a grbuffer as a texture in receiving a buffer item delivered from BQProducer. If n and n_{ex} of the buffer item are different, it means that n indicates a junk grbuffer number, so that it is necessary to display an exFrame grbuffer to be reused in scrolling, instead of a junk grbuffer. In such a situation, BQConsumer needs to designate an exFrame grbuffer indexed by n_{ex} as a texture, but has been designed to designate a grbuffer indexed by n as a texture, thus causing HWC to display an invalid frame if n is a junk grbuffer number. To make the exFrame grbuffer into a texture, SCAR substitutes the exFrame grbuffer handle for a junk grbuffer handle of mImages[n], which represents entry n in the mImages array and has the handle of a grbuffer to be designated as a texture.

Algorithm 2 describes the substitution policy implemented into a grbuffer acquiring operation, which BQConsumer executes upon the arrival of a buffer item to initialize the mapping between grbuffer numbers and grbuffer handles into the mImages array with three entries. This algorithm dequeues a buffer item with grbuffer number n from a buffer item queue (line 4). Since the creation of a layer or surface in a window, there is the arrival of the first three buffer items, each having n value of 0, 1, and 2 in order, and the handle of a grbuffer indexed by n. For each buffer item, an object called *Image* is created to store grbuffer and EGLImage handles for n, and is stored to mImage[n] (lines 8–11). However, the algorithm can change the grbuffer handle of the Image for n during the texture designation (line 18), thus backing up the grbuffer handle to keep track of its original value (line 12).

Algorithm 2. Algorithm of replacing a junk grbuffer handle of an mImages entry with an exFrame one

Input: Buffer item queue Q_{item} of storing a set of buffer items
Output: Buffer item *item*, array *mImages*
1 // assume that *Image* is a class with grbuffer handle *grbHandle* and EglImage handle *eglHandle*
2 // assume that *mImages* is an array of storing a set of *Image* objects
3 // assume that *grbOrigHandles* is an array of storing a set of original grbuffer handles
4 dequeue *item* from Q_{item}
5 $n \leftarrow$ grbuffer number of *item*
6 *grbHandle* $\leftarrow$ grbuffer handle of *item*
7 **if** *grbHandle* $\neq$ *null* **then** // The initialization of mImages
8 create *Image* object *img*
9 *img.grbHandle* $\leftarrow$ *grbHandle*
10 *img.eglHandle* $\leftarrow$ EglImage handle for n
11 *mImages*[n] $\leftarrow$ *img*
12 *grbOrigHandles*[n] $\leftarrow$ *grbHandle*
13 **else**
14 $n_{ex} \leftarrow$ exFrame grbuffer number of *item*
15 **if** $n = n_{ex}$ **then** // Original grbuffer handle is used for the texture designation
16 *mImages*[n].*grbHandle* $\leftarrow$ *grbOrigHandles*[n]
17 **else** // exFrame grbuffer handle is used for the texture designation
18 *mImages*[n].*grbHandle* $\leftarrow$ *mImages*[n_{ex}].*grbHandle*
19 **end**
20 **end**

Since the initialization of the mImages array, BQProducer delivers a buffer item with a *null* grbuffer handle, which is dequeued from a buffer item queue (line 14) to compare its n and n_{ex} (line 15). If they are different, the grbuffer handle of mImages[n] is replaced with that of mImages [n_{ex}] to designate an exFrame grbuffer as a texture (line 18) because n indicates a junk grbuffer number. If they are the same, n indices a grbuffer storing a new exFrame, but mImages[n] can have not the original handle of a grbuffer for n, but the handle of another grbuffer due to the replacement of the grbuffer handle in the entry. In such a situation, the original grbuffer handle for n is restored to the grbuffer handle of mImages[n] (line 16). With the buffer item, BQConsumer afterward designates a grbuffer

indexed by n as a texture by assigning mImages[n] to the mCurrentTextureImage, which is used to make the grbuffer into an EGLImage and bind the EGLImage to a texture.

4.5. Repositioning exFrame on Screen

Since the creation of a texture through an exFrame, SCAR repositions a texture on the screen by total scrolling distance d. If a screen in Figure 6a has the upper left corner (0, 0) and lower right corner (w, h), a texture has the upper left corner (0, 0) and lower right corner (w, $h \times N$). As described in Section 4.3, d indicates the Y-starting position where content to be displayed at the top of the screen is located within a texture. For the repositioning, SCAR specifies content between d and $d + h$ within the texture as target content, which is displayed to the screen region. The Android graphics pipeline displays a texture to a different location on the screen by changing its starting position in the mDrawingState with the layout information of a layer, but can reposition only the content currently displayed on the screen. As shown in Figure 6b, if it is attempted to display the target content in the texture to the screen region, only the screen content (i.e., 1 to 3) except for pre-rendered content is moved up, and the remaining region after the movement is filled with black instead of the pre-rendered content (i.e., 4).

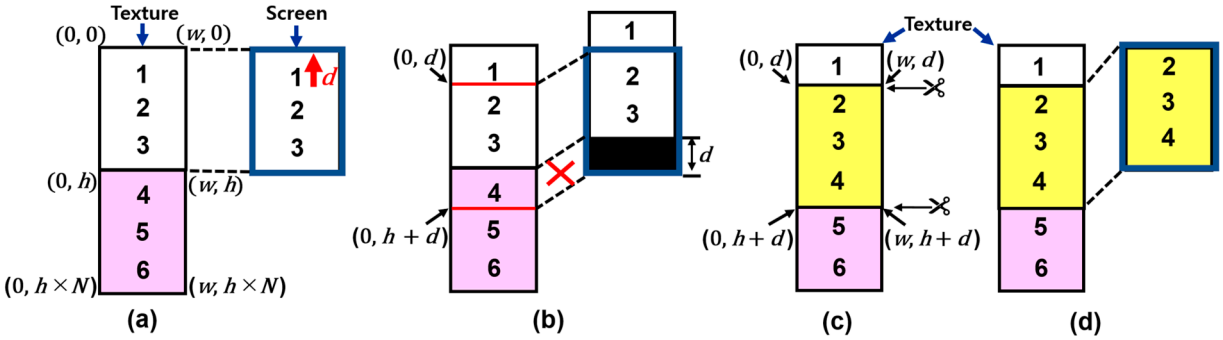

Figure 6. Cropping and mapping operations: (**a**) scrolling; (**b**) after scrolling; (**c**) cropping; (**d**) mapping.

To overcome the problem in Figure 6b, SCAR conducts the cropping and mapping operations implemented into the HWC setup routine, which initializes the geometry of a texture and informs HWC how to coordinate the texture on the screen. As shown in Figure 6c, the cropping operation first specifies the four lines of a rectangular region (yellow region) to be cropped in a texture. In this region, the left and right lines are indicated by 0 and w in width, respectively, and the top and bottom lines are indicated by d and $h + d$ in height, respectively. The cropping of the region is achieved by assigning the four line values into the data structure called *Crop*, which the Android defines in the mDrawingState for cropping a rectangular region. The mapping operation maps the starting position (0, d) of the cropped region into the starting position (0, 0) of the screen in the negative Y-direction to display the cropped region to the screen region, as shown in Figure 6d. This change is achieved by assigning $-d$ to the Y-starting position in the data structure called *transform*, which is defined in the mDrawingState for the purpose of moving a texture on the screen. Thereafter, HWC moves the texture by d in the Y-direction by using the mDrawingState when composing the texture, the navigation bar, and status bar into one image according to their Z-order levels.

4.6. Implementation Issues

There are some issues in implementing SCAR into the Android graphics pipeline due to the following architecture characteristics. First, an app consists of the UI thread in the Java layer and the render thread in the native layer. This separation of the two layers causes an issue in delivering a scrolling distance and the scrollability of a window from the UI thread to the render thread. Second, the three components, app, BQProducer, and SurfaceFlinger, run in separate processes. The non-open-source GLES library also interferes

in the flow control where the render thread enqueues a grbuffer with a rendered frame into a buffer queue. These characteristics made it difficult for us to deliver the scrollability and a scrolling distance from the UI thread to the render thread, and a total scrolling distance d and the rendering indicator from the render thread to BQProducer.

In SCAR, the scrollability and scrolling distance are added as member variables to class DecorView in the UI thread. Because its object is shared by several components, it serves as an intermediary to provide the two parameters for components cooperating to implement SCAR. The UI thread calls the JNI function nSyncAndDrawFrame() to request the render thread to render a frame. During this function call, we make the UI thread pass the two parameters of the DecorView to the function as added arguments, which are put into a rendering request called DrawFrameTask to the render thread. The render thread passes the parameters to the method draw() of the CanvasContext performing Algorithm 1 because the method needs them to decide whether to reuse an exFrame.

Figure 7 shows the process of how the render thread requests SurfaceFlinger to display a frame through the grbuffer swapping operation (line 22 in Algorithm 1), which is executed in the method SwapBuffers() of the Skia graphics engine. We made extensive efforts to investigate the process through a dynamic code analysis because the grbuffer swapping operation invokes the GLES API eglSwapBuffers() [15] whose source code is not public. It was discovered that this API invokes the method queueBuffer() of class Surface that requests BQProducer to dequeue/enqueue a grbuffer from/to a buffer queue. This method finds grbuffer number n by the handle of a grbuffer and executes a proxy method of BQProducer to transfer n via Binder IPC to BQProducer running on a process. The proxy method makes BQProducer execute the enqueue operation described in Section 4.4 by invoking its stub method queueBuffer() with n, which enqueues the grbuffer indexed by n into a buffer queue and sends a buffer item with n to SurfaceFlinger.

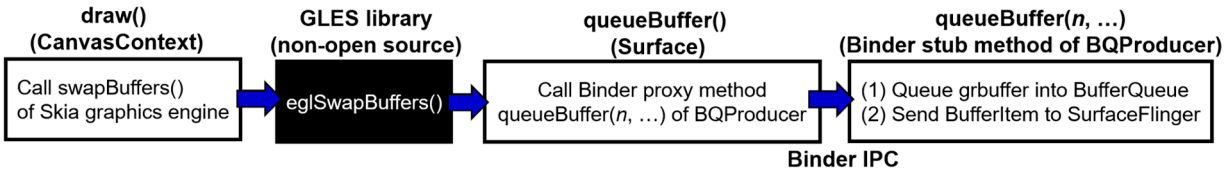

Figure 7. Process of requesting SurfaceFlinger to display an exFrame. These class and method names are the same across the various versions of the Android OS.

We implement the decision policy of exFrame grbuffer number n_{ex} mentioned in Section 4.4 into the stub method knowing the detail of the buffer queue. To send n_{ex} and d to SurfaceFlinger, the stub method needs to obtain the two important parameters, the rendering indicator and d, from the render thread executing Algorithm 1. However, there were two challenges in transferring the two parameters to the stub method via the Skia method swapBuffers() that triggers sending a buffer item to SurfaceFlinger. First, we cannot modify the source code of the eglSwapBuffers() to pass the parameters to its callee method queueBuffer() of the Surface because it is implemented in the GLES library. Second, a new Binder IPC channel should be created for transferring the parameters from the queueBuffer() to a process executing BQProducer. However, this creation requires a complete understanding of the Binder IPC and an Android rooting that can cause a system integrity problem.

The first challenge is solved using a design feature that class CanvasContext has a member variable referencing class Surface in the Android graphics pipeline. We provide the Surface with additional variables into which to store the two parameters, which the method draw() of the CanvasContext reflects into the variables in executing Algorithm 1. The queueBuffer() of the Surface retrieves the parameters from the variables of the Surface when it is invoked in the eglSwapBuffers(). The second challenge is solved using an argument packing scheme. When the queueBuffer() of the Surface calls the proxy method of BQProducer with a grbuffer number of 4 bytes as an argument, the scheme divides the

argument into three parts, which are packed with two bytes for a total scrolling distance, one byte for the rendering indicator, and one byte for a grbuffer number whose maximum is 64. The proxy method executes the stub method with the packed argument, the three parts of which are unpacked into the two parameters in the stub method, along with the grbuffer number. This scheme is simple to implement because it does not create a new Binder IPC channel and does not modify a Binder interface.

5. Experimental Evaluations

5.1. Experimental Environment

Our experiments were conducted on a Google Pixel 3XL smartphone running AOSP Android 9 (Pie), into which SCAR was implemented. Measurements were made using Qualcomm Snapdragon Profiler version 2020.4 [26], a tool for analyzing the power consumption and performance of smartphones with Snapdragon chipsets. The tool has been used in previous studies [28,29] to measure power consumption in mobile devices. We used a WiFi network in the experiments and minimized display brightness and turned off GPS and Bluetooth to eliminate unnecessary power consumption. Eight popular apps were selected for evaluation: Facebook, Twitter, and Tumblr (SNS), Hotels.com (hotel booking), News Republic (news), Twitch (video streaming), Wish (shopping), and Resplash (wallpaper). We chose these apps, first, because scrolling is a major use pattern in all the apps, and second, because they allow us to scroll for a long enough time to ensure a highly accurate measurement. In the apps, we scrolled for 10 min on scrollable windows.

We measured power saving over different exFrame sizes and scrolling speeds, each defined as scrolling distance per second (pixels/s). The exFrame size was configured as 1.5, 2, and 2.5 times the frame size of the Android graphics pipeline to examine which size best improves power consumption with minimal memory usage of a buffer queue. In a previous study [8], an investigation on major scrolling patterns revealed that users read content most while scrolling at a scrolling speed of approximately100 pixels/s and quickly turn over content without reading it at scrolling speeds exceeding 1000 pixels/s. Based on these patterns, we selected these two speeds as scrolling speeds for the evaluation and continued to scroll at a constant speed for each scrolling speed.

5.2. Frame Rate and Power Consumption

We conducted an experiment to evaluate frame rates over different scrolling speeds and exFrame sizes. Upon an up-scrolling operation, we measured the number of executions of the Skia rendering operation (line 20 in Algorithm 1) that requests the GPU to render a frame. The frame rate is calculated as the number of the Skia rendering operations for the total scrolling time. "Original" represents the Android graphics pipeline, and three SCARs, denoted as SCAR(1.5), SCAR(2), and SCAR(2.5), have exFrame sizes of 1.5, 2, and 2.5 times the frame size of "Original". In Table 2, "Original" renders frames at frame rates close to 60 fps, whereas all the SCARs render exFrames at frame rates of less than one fps because of reusing exFrames in scrolling. We observe a slight increase in frame rates with the increase of scrolling speed. A higher scrolling speed reduces the reuse time of an exFrame, thus causing more frequent up-scrolling overflows. In contrast, there is little impact of the exFrame size on the frame rate. The larger an exFrame, the longer its reuse time, but there is not a large decrease in frame rendering operations for the total scrolling time due to extremely low frame rates.

Figure 8 illustrates the impact of scrolling speed on power consumption. SCAR improves the power consumption by 9% to 28% at 100 pixels/s and by 14% to 30% at 1000 pixels/s compared to "Original". We observed 20% greater power saving in the five apps other than Wish, Twitch, and Tumblr for the speed of 100 pixels/s, and in the six apps other than Wish and Tumblr for the speed of 1000 pixels/s. This result indicates that extremely low frame rates have an important role in improving the power consumption in SCAR. As scrolling speed increases, "Original" incurs a higher increase in power consumption than SCAR in the five apps other than Hotels.com, Tumblr, and Resplash, where the increase is

almost the same in "Original". As a result, it is confirmed that SCAR can minimize power consumption in various apps, increasing in proportion to scrolling speed.

Table 2. Frame rate (fps) according to apps and scrolling speed.

App Name	Facebook		Twitter		Hotels.com		News Republic		Wish		Twitch		Tumblr		Resplash	
Scrolling speed (pixels/s)	100	1000	100	1000	100	1000	100	1000	100	1000	100	1000	100	1000	100	1000
Original	56	56	56	56	56	56	56	56	54	54	55	56	57	57	57	57
SCAR(1.5)	0.05	0.35	0.05	0.37	0.05	0.35	0.05	0.35	0.05	0.37	0.05	0.36	0.05	0.36	0.05	0.35
SCAR(2)	0.05	0.35	0.05	0.37	0.05	0.35	0.05	0.35	0.05	0.37	0.05	0.36	0.05	0.36	0.05	0.35
SCAR(2.5)	0.05	0.35	0.05	0.37	0.05	0.35	0.05	0.35	0.05	0.37	0.05	0.36	0.05	0.36	0.05	0.35

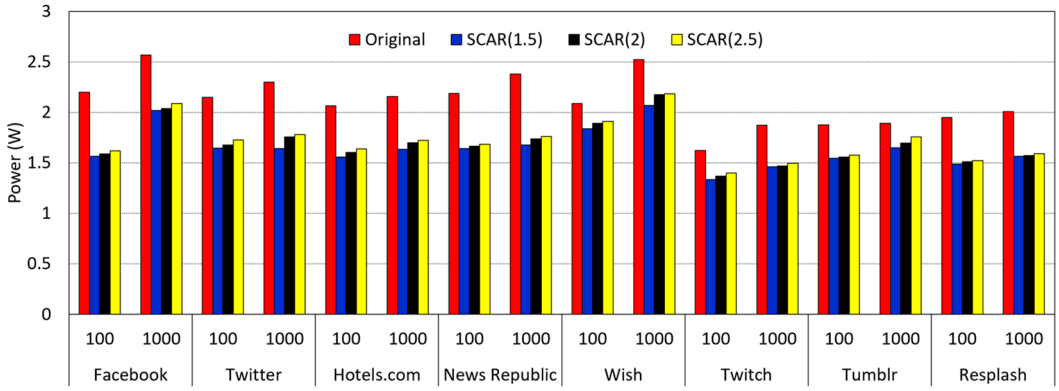

Figure 8. Power consumption for various apps and scrolling speeds.

We analyzed power consumption according to exFrame size in the apps that have the best performance on power savings. SCAR(1.5), SCAR(2), and SCAR(2.5) reduced power consumption by up to 28%, 27%, and 26% on Facebook; 24%, 22%, and 20% on Twitter; and 25%, 22%, and 20% on Hotels.com, respectively, for the speed of 100 pixels/s. There were 30%, 27%, and 26% reductions in the power consumption on News Republic, and 28%, 22%, and 20% on Twitter for the speed of 1000 pixels/s. It is notable that the increase in the exFrame size gives little opportunity to improve the power consumption due to extremely low frame rates. SCAR(1.5) has the best power savings in all the apps for the speed of 100 pixels/s and achieves better or almost the same power savings for the speed of 1000 pixels/s, compared to SCAR(2) and SCAR(2.5). Therefore, it is important to select an exFrame as small as possible if there is little difference in the power saving, because the memory usage of a buffer queue is linearly proportional to the exFrame size.

5.3. GPU and CPU Utilization

We examined the impact of scrolling on the average GPU utilization. As shown in Figure 9, SCAR reduces the GPU utilization by up to 100% and 99% for the speeds of 100 and 1000 pixels/s compared to "Original". Especially, there is a 100% reduction in the utilization on Tumblr and Resplash for the speed of 100 pixels/s. This result indicates that a significant decrease in frame rates can greatly improve the GPU overhead that is a major cause of the power consumption in the two apps. SCAR redraws as many views as the exFrame size increases, thus causing large GPU overhead in rendering the views. However, there is little impact from the exFrame size on the GPU utilization because frame rendering occurs very occasionally. Moreover, the GPU utilization varies greatly across apps in "Original", but has little difference among the apps in SCAR with very low GPU overhead. In contrast, the scrolling speed has a slight impact on the GPU utilization because scrolling overflows occur more frequently with the increase in the scrolling speed.

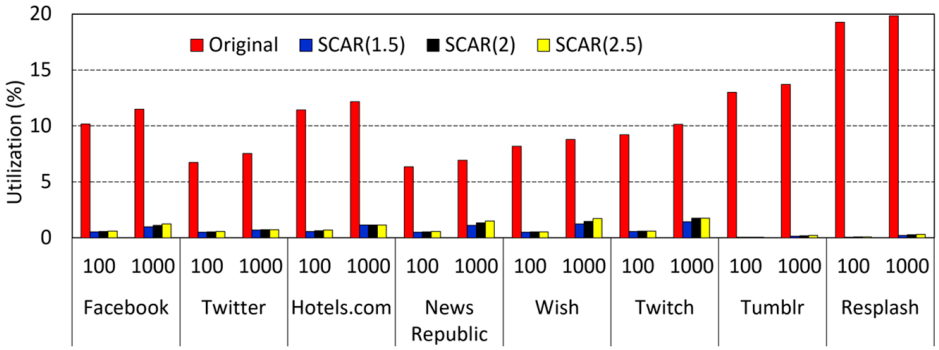

Figure 9. Average GPU utilization.

Figure 10 shows the variation in GPU utilization for a time interval when scrolling up on Hotels.com, with a great reduction in power consumption and GPU and CPU utilization. As shown in Figure 10a, the GPU utilization for the speed of 100 pixels/s remains at about 12% in "Original" due to consecutive rendering operations, whereas it remains level at about 0% in SCAR for the time intervals when an exFrame is reused, but rises up to 7%, 9%, and 11% in SCAR(1.5), SCAR(2), and SCAR(2.5), respectively, when an exFrame is rendered due to scrolling overflow. A larger exFrame size causes a slight increase in the GPU utilization due to the increase in the quantity of content to be rendered, but makes little impact on average GPU utilization due to extremely low frame rates, as shown in Figure 9. Figure 10b presents that there are a larger number of sudden increases in GPU utilization at the higher scrolling speed, which leads to each exFrame being reused for a shorter time. Such a reuse slightly increases the number of rendering operations.

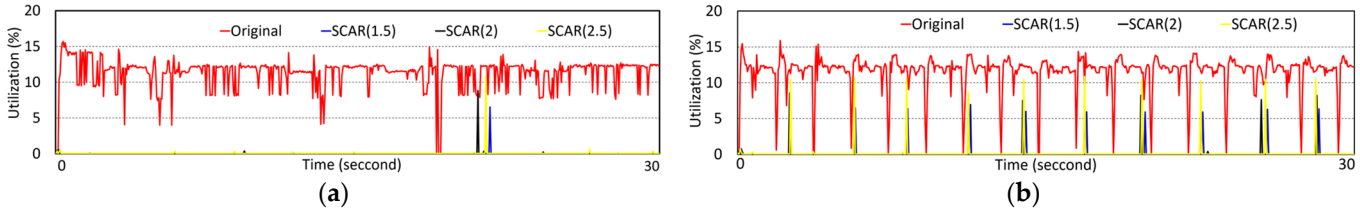

Figure 10. Variation in GPU utilization on Hotels.com: (**a**) scrolling speed of 100 pixels/s; (**b**) scrolling speed of 1000 pixels.

Figure 11 shows the average CPU utilization in scrolling operations. SCAR reduces the CPU utilization by 24% to 62% for the speed of 100 pixels/s and by 26% to 65% for 1000 pixels/s, because reusing an exFrame avoids translating drawing commands in display lists to GLES commands and calling these translated commands. This result confirms that the CPU utilization reduction contributes greatly to reducing power consumption. We present a detailed analysis of the CPU overhead in the next subsection to examine the root cause of the increases in CPU utilization.

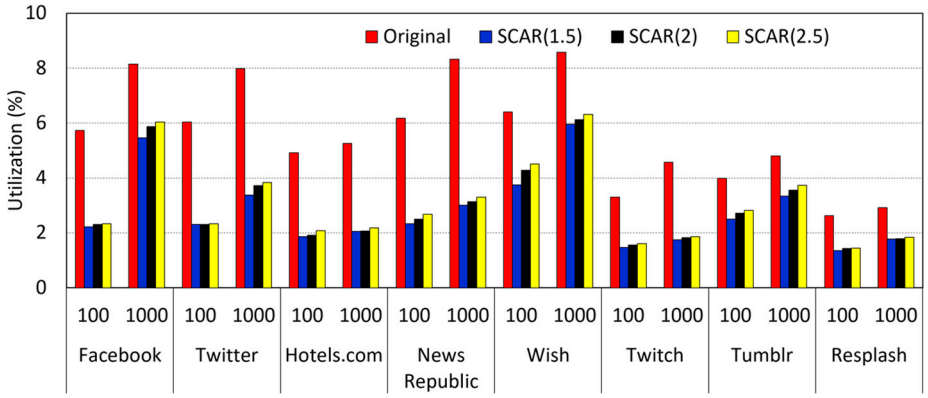

Figure 11. Average CPU utilization.

5.4. Analysis of CPU Overhead

Figure 12 shows the execution time of the method performTraversals() of class ViewRootImpl for the speed of 100 pixels/s, which searches views to be redrawn from a view tree and generates display lists for the found views. SCAR(1.5) increases the execution time by 4% on Facebook and by 31% to 60% in the other apps compared to "Original". The frame extension increases the view tree size or the number of views in a view tree, thus having the following negative impacts on the CPU overhead. The first is to increase the time spent in searching views to be redrawn from a view tree, and the second is to increase the time spent in generating display lists for views to be redrawn. Comparing Figure 12 and Table 1, we observe the increase in the execution time with increasing view tree sizes and increasing views to be redrawn. Facebook suffers from a larger increase in the execution time even if the app has a smaller view tree size and a smaller number of views to be redrawn compared to Twitch. The features of a view, such as size, shape, type, style, and complexity, can affect the number of drawing commands composing a display list for a view. If an app has a small number of views whose display lists have many drawing commands, a large CPU overhead can occur in updating the drawing commands into display lists to redraw the views.

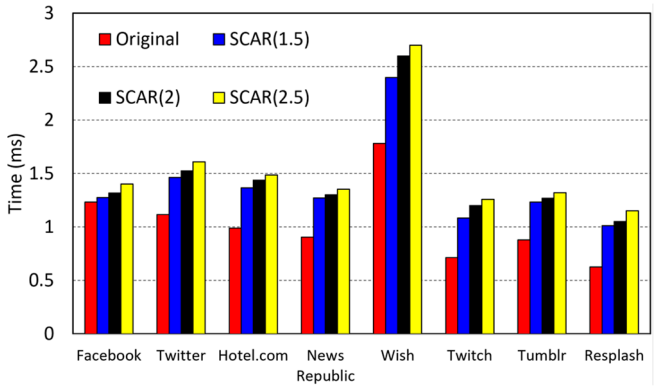

Figure 12. Execution time of the method PerformTraversals() of class ViewRootImpl.

Figure 13 shows the execution times of the method draw() of the CanvasContext for the speed of 100 pixels/s when "Original" and SCAR each invoke this method to render one frame and one exFrame, respectively. SCAR increases the execution time by 82% to 403% over "Original". Moreover, there is an increase in the execution time in proportion to the exFrame size in an app. To analyze these results, we measured the execution time of the Skia rendering operation (line 20 in Algorithm 1) executing the following operations, as shown in Figure 14. The first is the command translation (CT) operation to translate drawing commands in display lists to GLES commands. This translation causes a CPU

overhead because the CPU generates a complex set of GLES commands through each drawing command depending on the view feature. The second is the command execution (CE) operation to call the translated GLES commands and make the GPU render a frame.

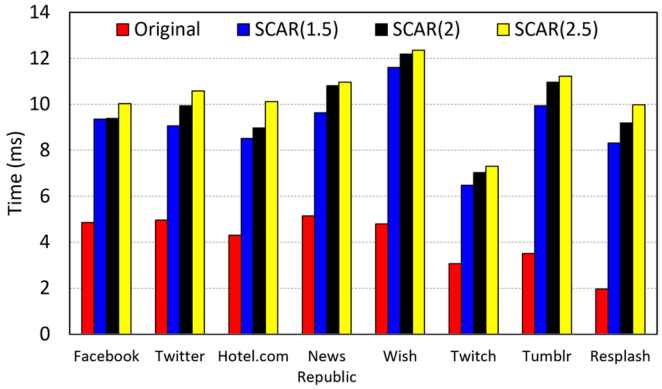

Figure 13. Execution time of the method draw() of class CanvasContext in a frame rendering operation.

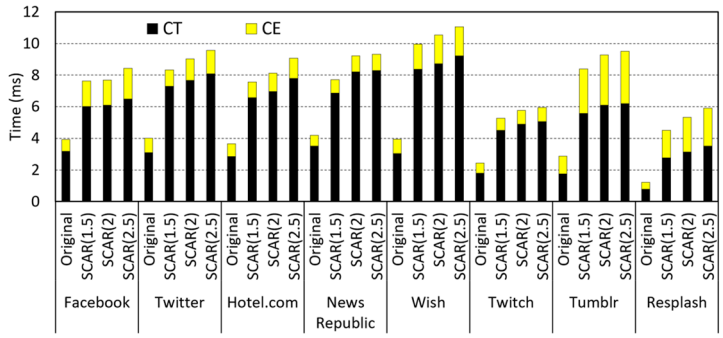

Figure 14. Execution time of the Skia rendering operation.

Comparing Figures 13 and 14, we observe that the Skia rendering operation spends more than 80% of the execution time of the draw() in all the apps. SCAR(1.5) increases the CT time by 89% to 252% compared to "Original". There is an increase in display lists because extending a frame causes more views to be redrawn. This increase makes the CPU translate more drawing commands to GLES commands. We observe an increase of 12% to 294% in the CE time in SCAR(1.5). As more GLES commands are generated, the CPU executes more GLES commands to render a larger frame, thus increasing the CPU overhead. The comparison of the CT and CE times indicates that the CT time accounts for a large portion of the overall execution of the Skia rendering operation, thus having a large impact on the CPU overhead. Moreover, it is notable that an app with fewer views can have a larger execution time of the Skia rendering operation than other apps. This results from the view features that have an impact on the number of drawing commands for a view.

We measured the effective execution time of the method draw() in which to render exFrames at the frame rates in Table 2, as shown in Figure 15 that includes the execution time for the "Original" method in Figure 13. The extremely low frame rates in SCAR indicate that the reusing operations of exFrames are dominant over the rendering operations executed by the Skia rendering operation with a large CPU overhead. This leads to the decrease of 67% to 86% in the execution time over "Original", because the Skia rendering operation is not executed in reusing an exFrame. This reduction surpasses the increase in the execution time of the performTraversals() in Figure 12, thus playing a major role in lessening the average CPU utilization, as presented in Figure 11. Moreover, there is little difference in the execution time of the draw() among the apps due to the dominant reusing

operations even if the Skia rendering operation has greatly different execution times across the apps.

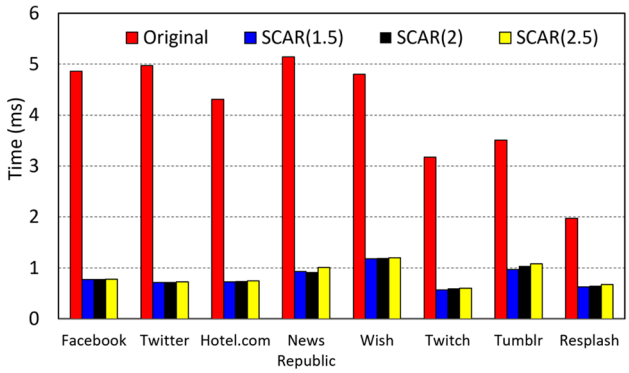

Figure 15. Overall execution time of the method draw() in a frame rendering operation.

The total rendering time taken to render a frame after a V-Sync signal should be less than 16.67 ms, due to the maximum frame rate of 60 fps; otherwise, there is a screen tearing problem where the frame cannot be updated to the screen. Figure 16 presents the total rendering time that is the sum of the two execution times of the performTraversals() and draw() for the speed of 100 pixels/s. The total rendering time is less than 16.7 ms in all the apps but increases in proportion to the exFrame size. As a result, SCAR(1.5) is the most appropriate, because it can not only avoid the screen tearing problem, but also save power, comparable to SCAR(2) and SCAR(2.5).

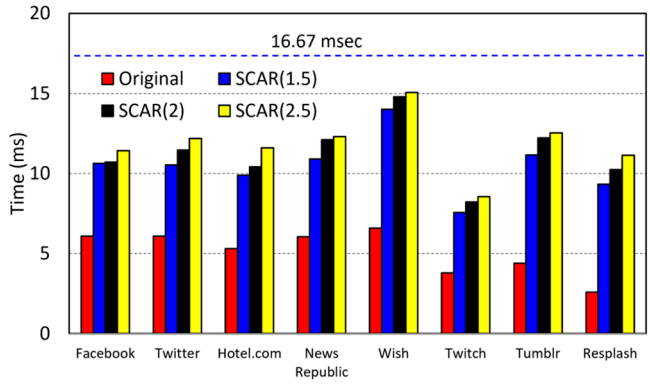

Figure 16. Total time spent to complete a frame rendering operation after a V-Sync signal.

5.5. Network Traffic and Window Launching Time

In SCAR, the pre-rendering can have a negative impact on the launching time of a scrollable window and the network traffic in scrolling on the window. The window launching time, which is an important user-perceived performance criterion, represents the time taken to display all the content of a window to a screen in launching the window for the first time. We measured the window launching time through the Android accessibility service API [30] that is used to assist users with disabilities in Android smartphones. The Snapdragon profiler is used to measure the network traffic that is the amount of data that an app transmits and receives with its remote server.

Figure 17 shows the network traffic in launching a scrollable window. SCAR(1.5) increases the network traffic by up to one fold, compared to "Original" respectively. There is also an increase in the network traffic in proportion to the exFrame size because a larger frame size causes an app to fetch more content from its remote server for the pre-rendering. Fortunately, SCAR slightly increases the network traffic on Twitter, Wish, and Resplash generating large network traffic. For example, SCAR(1.5) with the best power savings leads

to an increase of 1% to 5% in the network traffic on these three apps. On the other hand, the other apps with small network traffic suffer from an increase of 16% to 83% in the network traffic, but the increased amount of network traffic is very small. These results confirm that extending a frame does not cause large network traffic in launching a scrolling window.

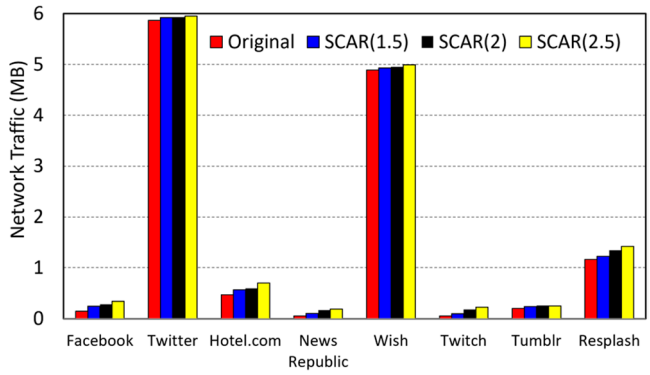

Figure 17. Network traffic upon the launching of a scrolling window.

Table 3 shows the network traffic per minute in scrolling on a scrollable window for the speed of 100 pixels/s after the launching of the window. In SCAR(1.5), there is little increase in the network traffic other than Hotels.com and Tumblr, in which the increase is small considering the scrolling time. This result is because the apps retrieve only content to be newly displayed to a user in scrolling. Hence, we confirm that SCAR has little impact on the increase in the network traffic in scrolling.

Table 3. Network traffic (bytes) per minute during scrolling.

App Name	Original (×1000)	SCAR(1.5) (×1000)	SCAR(2) (×1000)	SCAR(2.5) (×1000)
Facebook	126	127	127	128
Twitter	48	48	49	49
Hotels.com	284	300	380	405
News Republic	175	176	176	177
Wish	5651	5653	5658	5671
Twitch	324	325	329	354
Tumblr	580	618	635	652
Resplash	945	946	947	948

Figure 18 illustrates that SCAR(1.5) increases the window launching time by up to 3% over "Original", but the increase is nearly negligible in the five apps other than Wish, Twitch, and Tumblr. This result confirms that the pre-rendering has little impact on a user-perceived latency of launching a scrollable window. In SCAR(2.5), however, there is an increase of 3% to 24% in the window launching time due to the following two causes. The first is an increase in the total rendering time taken to render a frame, as shown in Figure 16. However, this increase has little impact on the window launching time that is much large compared to the total rendering time. The second is an increase in the network traffic upon launching a window whose size is dependent on a frame size. A larger frame size causes a launching window to fetch more content over the network for the pre-rendering, thus delaying when to start rendering the content. However, we expect that the high network speed can reduce the window launching time because of lessening the time taken to fetch content over the network.

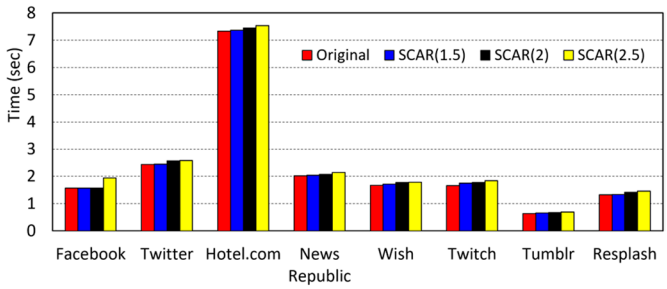

Figure 18. Window launching time in launching a scrollable window.

5.6. Power Consumption upon Down-Scrolling

We measured power consumption in scrolling down at a speed of 100 pixels/s to investigate whether a down-scrolling operation incurs unnecessary power consumption. Figure 19 indicates that there is little difference in the power consumption between SCAR and "Original" regardless of the exFrame size. As shown in Figure 5d, pre-rendering the content not reusable in a down-scrolling can cause unnecessary GPU and CPU overheads. To avoid this problem, SCAR renders render only views to be seen on screen like "Original" without pre-rendering views that cannot be reused in a subsequent down-scrolling, as shown in Figure 5e. This rendering behavior can avoid incurring unnecessary power consumption in a down-scrolling even though it does not benefit of reducing the power consumption through the pre-rendering. As a result, it is confirmed that SCAR saves significant power during up-scrolling without sacrificing the power consumption during down-scrolling.

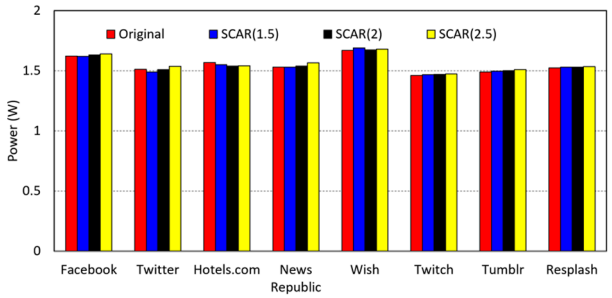

Figure 19. Power consumption in down-scrolling.

5.7. Power Consumption in Realistic Smarptphone Usage

We evaluated benefits of SCAR in realistic smartphone usage scenarios such as device setting and scrolling operations. First, the experimental smartphone was configured to have a realistic device setting by turning on Bluetooth, GPS and using a 50% display brightness, which is a typical indoor setting of a smartphone. Second, it is common that a user conducts scrolling after reading most of the displayed content. We designated the reading time intervals between scrolling operations as 0, 2, 5, and 10 s while scrolling at the speed of 100 pixels/s. A scrolling operation comes to stop for the reading time interval every time a user scrolls up from the bottom of a scrollable window to the top. The evaluation was conducted in SCAR(1.5) that leads the best power saving, network traffic, and window launching time.

Figure 20 shows the impact of the reading time interval on power consumption. In the realistic smartphone setting, SCAR(1.5) improves the power consumption by 9% to 23% at the reading time interval of 0 having no reading time, compared to "Original". This improvement is slight less than that in Figure 8 presenting the result in the experimental setting that turned off Bluetooth, GPS and minimized the display brightness. This is because the display panel, Bluetooth, and GPS persistently consume a specific amount of

power upon scrolling, thus offsetting the power consumption improved by SCAR(1.5). As the result, we confirm that SCAR has the advantage of saving the power even in a realistic operating scenario that uses a higher display brightness and turns on Bluetooth and GPS, compared to "Original".

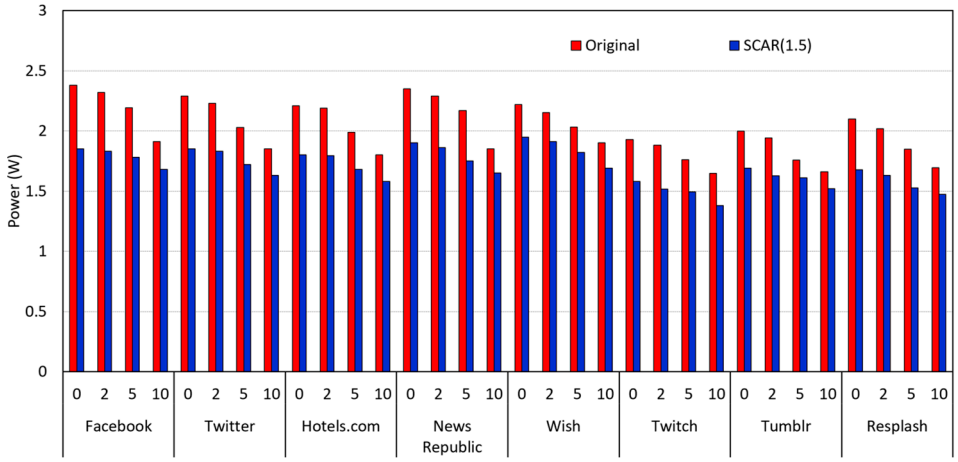

Figure 20. Power consumption for various apps in realistic smartphone usage scenarios considering a realistic device setting and reading time intervals.

As the reading interval time increases, there is a decrease in the power consumption in "Original" and SCAR(1.5) because the reading time interval reduces frame rates by lowering a frequency of scrolling operations. This decrease in the frame rates gives SCAR(1.5) less opportunity to improve the power consumption because SCAR(1.5) has a large advantage on scrolling operations causing high frame rates. Fortunately, SCAR(1.5) improves the power consumption by 9% to 21% and by 8% to 18% at the reading time intervals of 2 and 5 s respectively, compared to "Original". Moreover, there is the improvement of 7% to 15% even at the large reading time interval of 10 s. This result confirms that our scheme can improve the power consumption in the realistic smartphone usage scenario even if frame rates become lower due to the reading time interval.

6. Discussion

The implication for research is that our scheme has the advantage of improving power consumption in commercial apps by reducing frame rates upon scrolling in the Android graphics pipeline that has an important problem of rendering a frame at every scrolling movement. The implication for practice is that the core algorithms of our scheme are simple and effective because they do not incur additional CPU overhead in reusing an exFrame upon scrolling and are simply implemented in the Android graphics pipeline. Our research is efficiently applicable to real-world environments, but has the limitations in apps handling dynamic content and the recent mobile smartphones with a higher refresh rate.

Dynamic content can be displayed during scrolling: streaming or embedded videos might play, animated objects (e.g., progress status, GIFs, and emojis) might be updated, or pages of content might flip (or slide) left and right. There is an issue in reusing an exFrame with display dynamic content: during scrolling, the reusing repositions the stationary image of the dynamic content on screen, thus deteriorating the user experience. To solve this issue, SCAR detects the change in dynamic content to deactivate the reuse of an exFrame. This deactivation allows rendering a new exFrame during the change, similar to the process in the Android graphics pipeline.

SCAR cannot reduce frame rendering if dynamic content is played or updated upon scrolling, thus having little advantage of improving the power consumption. However, there is a benefit in saving the power upon scrolling a scrollable window where dynamic content such as embedded video and progress status is frozen without the update. We

inquired into 80 popular apps from 13 categories in the Korean Google Play Store to evaluate how many apps display dynamic content during scrolling. This inquiry was conducted on a main window appearing when the app launches. We found that 45 apps display no dynamic content, and 27 apps present a ViewPager for page flipping once at the launch but do not display dynamic window content after the view is passed by scrolling. In contrast, eight apps automatically play streaming videos during scrolling but provide a setting to disable the automatic play. A user can disable automatic play to save mobile network usage. This result of the inquiry indicates that our scheme is highly applicable to a significant percentage of all apps for reducing frame rendering.

SCAR might cause a screen tearing in the recent smartphones with the high refresh rate of 120 Hz due to the increase in the total rendering time. The total rendering time taken to render a frame should be less than 8.33 ms because a V-Sync signal occurs every 8.33 ms in the high refresh rate, but is larger than 8.33 ms in some apps. Fortunately, the recent smartphones with the 120 Hz refresh rate have become equipped with much higher performance CPUs than those with the 60 Hz refresh rate [31]. This is because the high refresh rate needs a powerful processor to make games come alive without a screen tearing, make animations smoother, and make high-resolution videos played at the frame rate of 120 fps [32]. For example, the Galaxy S21 series and the OnePlus 9R have two times higher CPU performance than that of our experimental smartphone. Thus, we expect that the recent powerful processors make it possible for SCAR to work in advanced mobile environments efficiently. Furthermore, the Android graphics pipeline conducts more frequent rendering at a higher refresh rate upon scrolling, thus causing a larger power consumption. SCAR significantly reduces frame rates even at a higher refresh rate because it does not conduct a frame rendering until the ending of an exFrame reaches above the screen bottom upon scrolling. Thus, we expect that there is a wider gap of the power consumption between the Android graphics pipeline and SCAR in the recent smartphone supporting the high refresh rate of 120 Hz.

Currently, we cannot evaluate SCAR in smartphones with the 120 Hz refresh rate due to the following two reasons. First, the Google has not still released a reference phone supporting the high refresh rate. We need the reference phone to implement SCAR into the Android graphics pipeline because only a reference phone can run the Android OS with modified source code. Second, it takes a long time for the Google to release low-level software such as device drivers and firmware even if the reference phone is launched in the near feature. In the future, we will verify SCAR on a reference phone with the 120 Hz refresh rate and optimize its performance.

The GUI development practice and cross-platform app development can have an impact on power consumption in the user interaction with an app. Some studies [33–35] introduced that the design and implementation of UI element (i.e., view) layout are important factors in UI performance and power consumption. The more complex view hierarchy (or tree structure) and higher UI overdraw indicating that the same pixels are redrawn for multiple times in a frame rendering, the larger the CPU and GPU overheads of the Android graphics pipeline. As shown in Table 1, the Android graphics pipeline redraws many of views displayed most recently, upon scrolling. In such a behavior, we expect that the Android graphics pipeline can incur a larger power consumption upon scrolling if a scrollable window consists of a view layout with more complex view hierarchy and higher UI overdraw. In future work, we will investigate the influence on power consumption of the GUI development practices, including a method of minimizing redrawing operations of views displayed most recently upon scrolling. Moreover, a study [36] introduced that the cross-platform app development such as React and Ionic causes a performance penalty due to unoptimized frame rendering and redrawing operations in some simple UIs, compared to native Android and iOS apps. However, the study did not investigate power consumption in common UI designs and use patterns such as scrolling. We confirm that the cross-platform frameworks have an impact on power consumption upon scrolling, thus planning to scrutinize the impact as the further work.

7. Conclusions

This study showed that scrolling makes the Android graphics pipeline render most or all of the previously rendered content at a frame rate of nearly 60 fps. Our scheme, called SCAR, when rendering a frame for content to be displayed, prerenders any content that is likely to be displayed soon in a subsequent scrolling. In the subsequent scrolling, SCAR reuses the most recently rendered frame instead of rendering a new frame. We implemented SCAR into the complicated Android smartphone environment to evaluate the power savings. Our evaluation demonstrated the efficiency of reducing frame rates and reducing power consumption. As a result, we expect that SCAR will be applied into the Android graphics pipeline in the near future.

Author Contributions: Conceptualization, W.-H.A. and S.-H.L.; methodology, W.-H.A., C.-K.H. and S.-H.L.; software, W.-H.A., C.-K.H., K.-M.H. and S.-H.C.; validation, W.-H.A., C.-K.H., K.-M.H. and S.-H.C.; formal analysis, W.-H.A. and J.-W.O.; investigation, W.-H.A. and C.-K.H.; resources, W.-H.A. and J.-W.O.; data curation, W.-H.A. and C.-K.H.; writing—original draft preparation, W.-H.A. and S.-H.L.; writing—review and editing, S.-H.L. and J.-W.O.; visualization, W.-H.A.; supervision, S.-H.L.; project administration, W.-H.A.; funding acquisition, W.-H.A., S.-H.L. and J.-W.O.; All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2019R1F1A1062308, NRF-2021R1F1A1048026, NRF-2021R1F1A1048842). The work reported in this paper was conducted during the sabbatical year of Kwangwoon University in 2019. This work was supported by Hankuk University of Foreign Studies Research Fund.

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Carroll, A.; Heiser, G. An analysis of power consumption in a smartphone. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 23–25 June 2010; pp. 21–34.
2. Chen, X.; Chen, Y.; Dong, M.; Zhang, C. Demystifying energy usage in smartphones. In Proceedings of the Design Automation Conference, San Francisco, CA, USA, 1–5 June 2014; pp. 1–5.
3. Anand, B.; Thirugnanam, K.; Sebastian, J.; Kannan, P.; Ananda, A.; Chan, M.; Balan, R. Adaptive display power management for mobile games. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys), Bethesda, MD, USA, 28 June–1 July 2011; pp. 154–196.
4. Chen, X.; Zheng, J.; Chen, Y.; Zhao, M.; Xue, C.J. Quality-retaining OLED dynamic voltage scaling for video streaming applications on mobile devices. In Proceedings of the Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012; pp. 1000–1005.
5. Lee, B.H.; Kim, Y.J. Dynamic voltage scaling using scene change detection for video playback on mobile AMOLED displays. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), San Francisco, CA, USA, 8–10 August 2016; pp. 302–307.
6. Dong, M.; Choi, Y.; Zhong, L. Power-saving color transformation of mobile graphical user interfaces on OLED-based displays. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), San Francisco, CA, USA, 19–21 August 2009; pp. 339–342.
7. Lee, G.; Lee, S.; Kim, G.; Choi, Y.; Ha, R.; Cha, H. Improving energy efficiency of android devices by preventing redundant frame generation. *IEEE Trans. Mob. Comput.* **2019**, *18*, 871–884. [CrossRef]
8. Yu, J.; Han, H.; Zhu, H.; Chen, Y.; Yang, J.; Zhu, Y.; Xue, G.; Li, M. Sensing human-screen interaction for energy-efficient frame rate adaptation on smartphones. *IEEE Trans. Mob. Comput.* **2015**, *14*, 1698–1711. [CrossRef]
9. Seo, B.; Kim, H.; Kim, S. Freezing: Eliminating unnecessary drawing computation for low power. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 56–61. [CrossRef]
10. Google Inc. Graphics–Android Open Source Project. Available online: https://source.android.com/devices/graphics (accessed on 1 July 2021).
11. Kim, D.; Jung, N.; Chon, Y.; Cha, H. Content-centric energy management of mobile displays. *IEEE Trans. Mob. Comput.* **2016**, *15*, 1925–1938. [CrossRef]
12. Kushner, D. The video game software wizardry of Id. *IEEE Spectr.* **2002**, *39*, 42–47. [CrossRef]
13. Google Inc. Android Open Source Project. Available online: https://source.android.com (accessed on 1 July 2021).

14. Khronos Group. OpenGL ES Overview. Available online: https://www.khronos.org/opengles (accessed on 1 July 2021).
15. Khronos Group. Khronos Native Platform Graphics Interface. Available online: https://www.khronos.org/registry/EGL/specs/eglspec.1.5.pdf (accessed on 1 July 2021).
16. Intel Developer Zone. Using OpenGL ES to Accelerate Apps with Legacy 2D GUIs. Available online: https://software.intel.com/en-us/articles/using-opengl-es-to-accelerate-apps-with-legacy-2d-guis (accessed on 1 July 2021).
17. Tan, K.; Okoshi, T.; Misra, A.; Balan, R. Focus: A usable and effective approach to OLED display power management. In Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous computing (UbiComp), Zurich, Switzerland, 8–12 September 2013; pp. 573–582.
18. Hwang, C.; Pushp, S.; Koh, C.; Yoon, J.; Liu, Y.; Choi, S.; Song, J. RAVEN: Perception aware optimization of power consumption for mobile games. In Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking, Snowbird, UT, USA, 16–20 October 2017; pp. 422–434.
19. Egilmez, B.; Schuchhardt, M.; Memik, G.; Ayoub, R.; Soundararajan, N.; Kishinevsky, M. User-aware frame rate management in android smartphones. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 1–17. [CrossRef]
20. He, S.; Liu, Y.; Zhou, H. Optimizing smartphone power consumption through dynamic resolution scaling. In Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, Paris, France, 7–11 September 2015; pp. 27–39.
21. Nixon, K.; Chen, X.; Zhou, H.; Liu, Y.; Chen, Y. Mobile GPU power consumption reduction via dynamic resolution and frame rate scaling. In Proceedings of the 6th Workshop on Power-Aware Computing and Systems (HotPower), Broomfield, CO, USA, 5 October 2014; pp. 5–9.
22. Pathania, A.; Jiao, Q.; Prakash, A.; Mitra, T. Integrated CPU-GPU power management for 3D mobile games. In Proceedings of the Design Automation Conference, San Francisco, CA, USA, 1–5 June 2014; pp. 1–6.
23. Pathania, A.; Irimiea, A.E.; Prakash, A.; Mitra, T. Power-performance modeling of mobile gaming workloads on heterogeneous MPSoCs. In Proceedings of the Design Automation Conference, San Francisco, CA, USA, 7–11 June 2015; pp. 1–6.
24. Chen, W.; Cheng, S.; Hsiu, P.; Kuo, T. A user-centric CPU-GPU governing framework for 3D games on mobile devices. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Austin, TX, USA, 7–10 November 2016; pp. 224–231.
25. Apple Developer. Prefetching Collection View Data. Available online: https://developer.apple.com/documentation/uikit/uicollectionviewdatasourceprefetching/prefetching_collection_view_data (accessed on 1 July 2021).
26. Qualcomm Technologies Inc. Qualcomm Snapdragon Profiler. Available online: https://developer.qualcomm.com/software/snapdragon-profiler (accessed on 1 July 2021).
27. Skia Inc. Skia Graphics Library. Available online: http://skia.org (accessed on 1 July 2021).
28. Xia, C.; Zhao, J.; Cui, H.; Feng, X.; Xue, J. DNNTune: Automatic benchmarking DNN models for mobile-cloud computing. *ACM Trans. Archit. Code Optim.* **2020**, *16*, 1544–3566. [CrossRef]
29. Yun, J.; Lee, J.; Kim, C.G.; Lim, Y.; Nah, J.H.; Kim, Y.; Park, W.C. A practically applicable performance prediction model based on capabilities of texture mapping units for mobile GPUs. *IEEE Access* **2019**, *7*, 102975–102984. [CrossRef]
30. Google Developers. Android AccessibilityService. Available online: https://developer.android.com/reference/android/accessibilityservice/AccessibilityService (accessed on 1 July 2021).
31. GeekBench Browser. Android Benchmarks. Available online: https://browser.geekbench.com/android-benchmarks (accessed on 22 August 2021).
32. Samsung Exynos. Mobile Processor Exynos 990. Available online: https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-990 (accessed on 22 August 2021).
33. Sillars, D. *High Performance Android Apps*; O'Reilly Media: Sebastopol, CA, USA, 2015; pp. 73–119.
34. Ding, N.; Hu, Y.C. GfxDoctor: A holistic graphics energy profiler for mobile devices. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys), Belgrade, Serbia, 23 April 2017; pp. 359–373.
35. Yan, Y.; He, S.; Liu, Y.; Huang, L. Optimizing power consumption of mobile games. In Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower), Monterey, CA, USA, 4 October 2015; pp. 21–25.
36. Biørn-Hansen, A.; Grønli, T.M.; Ghinea, G. Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance. *Sensors* **2019**, *19*, 2081. [CrossRef]