FlexDisplay: A Flexible Display Framework To Conserve Smartphone Battery Power

Anshak Goel*, Deeptorshi Mondal*, Manavjeet Singh†, Sahil Goyal*,
Navneet Agarwal*, Jian Xu†, Mukulika Maity*, Arani Bhattacharya*
*{anshak20283, deeptorshi20294, sahil20326, navneet18348, mukulika, arani}@iiitd.ac.in
†{manavsingh, jianxu1}@cs.stonybrook.edu
*IIIT-Delhi, †Stony Brook University

***Abstract*—Despite significant improvements, smartphones are still constrained by the limited capacity of their batteries. Modern smartphones tend to use organic light-emitting diode (OLED) displays, whose energy consumption depends both on the brightness and the color content. Since the display of smartphones is known to consume a significant portion of this energy, a number of prior systems have tried to reduce screen brightness, increase areas of dark zones on the screen or use colors that consume less energy to mitigate this problem. However, the amount of energy savings using these techniques are still limited, as the underlying compute required to render the content still consumes energy. In this work, we provide a framework FlexDisplay that disables the display of a limited portion of the app content, saves the underlying compute needed to render the content as well as the touch sensors in the corresponding display area. FlexDisplay supports disabling of content across multiple apps. We demonstrate it on 15 apps over different genres and show that the energy savings vary from 10%–47% of the total energy consumption of the smartphone, depending on the app and the disabled content.**

I. Introduction

The quality of smartphone hardware, such as compute capability of processors and size of display screen, has increased significantly over the last decade. However, the capacity of batteries has not increased at the same rate. Thus, smartphones are still energy-constrained. A number of studies have shown that high energy consumption has the strongest negative impact on the quality of experience of users [1]. Thus, it is essential to find additional ways of conserving battery energy.

A significant proportion of the power consumption comes from the display of the smartphone. A number of prior studies have proposed changing the content displayed on smartphone screens. For example, [2] uses a dark mode that disables colors from the screen. Focus [3] darkens a part of the screen that is less important to the experience of users to save power. Yet another set of works tend to either utilize colors that consume less energy to display or reduce the brightness of the screen [4]. As discussed by [2], the power savings obtained using these techniques is relatively moderate ($< 10\%$ of the total power). This is because these techniques target reducing only the display power, which only forms a relatively small portion of the total power in modern smartphones. The limited savings is due to the fact that these techniques cannot reduce the computation involved in rendering the content.

Our approach towards energy savings is motivated by a few observations. First, today's apps do not allow users to disable parts of the user interface (UI) even if they are not needed. For example, video conferencing apps like Google Meet and Skype allow users to stop their own camera, but do not allow users to stop rendering the video feeds of other users. Second, although screen darkening does save some energy, it does not achieve its full potential in terms of power saving as the content continues to be rendered. Thus, the CPU and the GPU of the system-on-chip continues to consume the same power as before. Thus, *an approach that combines disabling parts of the UI along with the compute needed to render them on the screen* is needed to further save power.

In this work, we design a framework called FlexDisplay that leverages these insights to enable selective rendering of UI elements. The first challenge is to implement this selective rendering design. This selective rendering is designed to ensure that disabled components do not need to be processed by the smartphone's system-on-chip (SoC) CPU. One notable aspect of FlexDisplay is its ability to provide end-users with options to disable specific parts of the content based on their preferences. For instance, users can choose to disable the video playback window while keeping the audio active. This allows users to customize their content viewing experience, conserving power by displaying only the essential information that is relevant to them.

A second challenge handled by FlexDisplay is that disabling a part of the display creates a number of problems related to usability. When a portion of the display is disabled, users may inadvertently tap on the disabled area, triggering actions that are not visible or expected. This can lead to confusion and disrupts the user experience. To overcome this challenge, FlexDisplay incorporates changes to the Linux kernel. Specifically, it modifies the kernel to ignore any touch inputs received on the disabled portion of the screen. By disregarding touch events within the disabled region, FlexDisplay prevents mis-tapping by users and eliminates the possibility of unintended actions being triggered. In addition to mitigating mis-tapping issues, this approach also contributes to energy savings. By ignoring touch inputs in disabled areas, FlexDisplay reduces the unnecessary processing and power consumption associated with handling unintended touches.

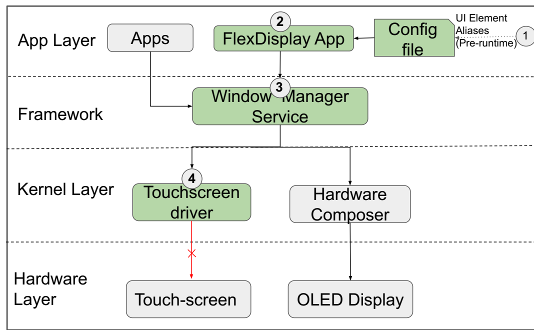

Fig. 1: Architecture of FlexDisplay in the context of the entire Android software stack. The blocks in green are the ones that FlexDisplay has added or modified.

By utilizing these techniques, FlexDisplay has the potential to generalize power-saving techniques to a large number of apps across various categories. This approach streamlines the process, reduces the burden on app developers, and empowers users to customize their power-saving settings according to their specific needs and preferences. We evaluate the power savings using FlexDisplay. Our evaluation includes apps across a wide range of categories, such as news, shopping, video streaming, video conferencing, camera, sport, social networking, and podcast. We obtain a power saving of 10.11% – 47.47% (median: 14.59%) of the total power consumption, with the smallest and largest savings coming from the Wall Street Journal (news category) app and Skype (video conferencing category) respectively. This shows that FlexDisplay drastically saves power and prolongs battery life.

II. Overview of FlexDisplay

FlexDisplay is designed in four major steps as shown in Fig. 1. The first step involves associating the UI elements visible to the user on the display with their corresponding class names. This association helps in identifying and referencing specific UI components. The second step entails obtaining user feedback on their comfort level in disabling parts of the UI. Users are given the option to specify which parts of the UI they are comfortable with disabling. The outcome of these two steps is the generation of a configuration file. In the third step, the configuration file is utilized by the modified View System to determine whether to render a particular UI component or not. This decision is based on the user's preferences outlined in the configuration file. In the fourth step, the configuration file is once again utilized, but this time to disable touch input on the corresponding portions where rendering has been disabled. This is achieved by modifying a kernel module responsible for handling the touchscreen in the hardware.

To finish all these steps, FlexDisplay upgrades multiple layers of mobile operating system (Fig. 1). That includes OS Framework Layer, to include the modifications to the View System within the Android Framework; Hardware Abstraction Layer, to disable mis-tapping via changes to touchscreen driver module. Finally, it makes additional changes to the application layer, as FlexDisplay also offers an application app for users to selectively choose which UI to disable.

III. Implementation of FlexDisplay

We now discuss in detail the implementation of each of the four steps of FlexDisplay.

1. Building Naming Aliases of UI Elements: The first challenge of FlexDisplay is to enable intuitive specification of the UI elements that could be disabled. We achieve this by identifying all the class files corresponding to the UI elements by using a tool LayoutInspector [5], which is packaged with Android Studio. The LayoutInspector provides an interface where a single click on any UI element shows the corresponding class. Note that this tool does not need access to the source code, thus enabling its use on all apps on the play store.

2. Customizing UI for Partial Display: It is in general not feasible for a user to directly modify the class methods, because that requires significant programming background and context. Thus, we have developed a FlexDisplay app that empowers users with the ability to toggle the inclusion or exclusion of images or videos for any app. This specialized FlexDisplay app examines the configuration file and allows users to easily enable or disable images and videos for each individual app according to their preferences. By presenting a user-friendly interface, the app simplifies the customization process, offering a straightforward way for users to indicate their desired settings.

3. Updating and Reading of Configuration: FlexDisplay then stores the updated configuration in a common directory on the local SD card of the device. The decision to store the configuration file in a common directory rather than individual app directories is primarily driven by Android's security restrictions. Android imposes limitations on accessing app directories from third-party apps to ensure data privacy and maintain the security of each app's data. By storing the configuration file in a common directory, the system creates a bridge that facilitates the transfer of user configurations across different apps. The next two components can then load it as a special file within the privileged file system group, to identify the UI elements and regions that need to be disabled.

4. Disable UI Rendering: After a user has completed the UI customization, FlexDisplay would disable the dedicated UI rendering for power saving. We add a separate thread to the View class to read the config file once every second, and identify if rendering of any UI element needs to be disabled. The *View* class stands for the basic unit for each UI element, such as a button, a image, or a video frame, etc. Each View class has function calls that are responsible for rendering the UI. For example, *invalidate*, *draw*, and *updateDisplayListIfDirty* functions. Consequently, these function calls are specifically designed to handle the rendering of respective UI elements.

In order to disable UI rendering at the View level, we introduce a new class named *DisableView* within the Android framework. This DisableView class overrides the existing versions of the invalidate(), draw(), and updateDisplayListIfDirty() function calls. The key difference is that the DisableView versions of these function calls include an additional check to determine whether the user or developer has disabled

App Category	Disabled Part(s)	App Name	Original Power (W)	FlexDisplay Power (W)	FlexDisplay Saving (%)
News	Images in articles	Reuters	1.282	1.062	17.16
		Wall Street Journal	1.68	1.51	10.12
		Google News	1.45	1.294	10.76
		InShorts	1.904	1.686	11.45
Shopping	Preview images of products	Zomato	1.816	1.576	13.22
		Ebay	1.588	1.386	12.72
Video Conferencing	Video of other participants	Google Meet	4.65	2.48	46.67
		Skype	3.674	1.93	47.47
Video Streaming	Parts of the screen that streams or runs the video	Youtube	1.494	1.276	14.59
		HotStar	1.82	1.562	14.18
Camera	Video preview	Camera	4.37	3.014	31.03
		OpenCamera	5.4	3.822	29.22
Sport	Images of articles & videos of discussions	Cricbuzz	1.318	1.172	11.08
Social Network	Profile photo & Images shared	Twitter	1.326	1.086	18.1
Podcast	Album cover for songs	Spotify	0.952	0.738	22.48

TABLE I: A list of apps, their categories, its display parts disabled, and power savings obtained using FlexDisplay.

UI rendering through the configuration file. If UI rendering has been disabled, the DisableView class immediately returns without proceeding to render the UI. On the other hand, if UI rendering is not disabled, the View class calls the original methods defined within it to render the UI elements as usual. Since the View class is applicable to all UI elements, introducing the DisableView class into the View class ensures that this solution scales and applies to all UI elements across all apps. Hence, the UI render can be automatically disabled once the user completes the UI customization.

5. Disable UI touching interaction: Once the rendering of a specific UI element is disabled, the UI element will no longer be rendered or displayed on the screen. However, this can result in the app having blank space, which may confuse users if they accidentally click/scroll/tap on it and find it still responsive. Therefore, it is necessary to disable the touch interaction for those corresponding invisible UI elements. By disabling the touch functionality, users will no longer be able to interact with the empty space, thereby avoiding any confusion or unintended actions.

Note that such changes in the device driver of kernel is device-specific, i.e. it comes at the cost of portability. This is because each device driver has its own way of handling touch. In our implementation on the Google Nexus 6P phone, we utilized the Synaptics touch module [6]. Although the part of the code where we check the touching region is device-specific, we expect the touch modules of other smartphones to be very similar in nature as the information from the upper layers come in the same form into all touchscreen drivers.

IV. Evaluation

We first discuss the power savings using FlexDisplay. Table 1 shows a total of 15 apps across 7 different categories. We use a Monsoon Power Monitor [7] to measure the power consumption. Since power fluctuates over time, we run the app, take the measurements for two minutes, and take the average over these two minutes. Further, we take measurements in five such iterations and report the mean power savings. We performed all the experiments on a Google Nexus 6 smartphone. Its display is of size 5.96 inches, 97.9 cm2, with resolution of 1440×2560 pixels and $16 : 9$ ratio. It has Qualcomm APQ8084 Snapdragon 805 (28 nm) chipset with CPU as Quad-core 2.7 GHz Krait 450 and GPU as Adreno 420.

The last three columns of Table 1 further shows the power savings obtained on each app. We note that the amount of power saved exceeds 10% in each of the apps. Note that this power saving is based on comparison of the total power, even though FlexDisplay optimizes only the display stack. Furthermore, the amount of power saved is especially high (exceeding 30%) for the camera and video conferencing apps, which are most power-intensive. This is because playing videos consume higher power. We conclude that FlexDisplay can save significant amount of power, thus showing its potential to prolong the battery lives of smartphones.

We also show the impact of disabling UI elements on a total of three representative apps – Spotify, Youtube and Zomato. Fig. 2 shows the screenshots with both FlexDisplay enabled and disabled. We note that in both Spotify and Zomato, it is still possible to access all the UI elements and interact as usual. This indicates that for such cases, it is possible to still utilize these apps with all its functionalities. For Youtube, only audio is available, and video is disabled, but all the other controls are available. This makes it ideal in case a user plans to utilize it for listening to music.

V. Related Work

A number of prior works try to save energy by optimizing the display of smartphones. For example, [8] and [9] both vary refresh rates on the display, depending on the content. The rise of OLED displays allowed additional optimizations, as darkening of pixels of them save significant amount of energy [2]. For example, brightness dimming [10], [11] is available on almost all modern smartphones to reduce energy consumption. ULPM [12] extends it by allowing users to interact without displaying any content on the smartphone display. Focus [3] reduces the brightness based on the importance of different content to users. FingerShadow [13] darkens pixels close to the user's fingers during their interaction as these pixels are not perceived by the users. ShutPix [14] develops a library that can reduce the density of pixels of some apps. FlexDisplay utilizes similar strategies, but goes further than these studies by also disabling the rendering to save significantly more energy.