FlexDisplay: An Optimized Smartphone Display Framework To Conserve Battery Power

ANSHAK GOEL*, IIIT-Delhi , India
DEEPTORSHI MONDAL*, IIIT-Delhi, India
MANAVJEET SINGH*, IIIT-Delhi, India
SAHIL GOYAL, IIIT-Delhi, India
NAVNEET AGARWAL, IIIT-Delhi, India
JIAN XU, Stony Brook University, USA
MUKULIKA MAITY, IIIT-Delhi, India
ARANI BHATTACHARYA, IIIT-Delhi, India

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 implement FlexDisplay on two different smartphones. 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. Furthermore, we show via user studies on 20 users that the changes made by FlexDisplay do not hurt their experience significantly.

CCS Concepts: • **Hardware → Displays and imagers**; **Power estimation and optimization**; • **Human-centered computing → Ubiquitous and mobile computing systems and tools**.

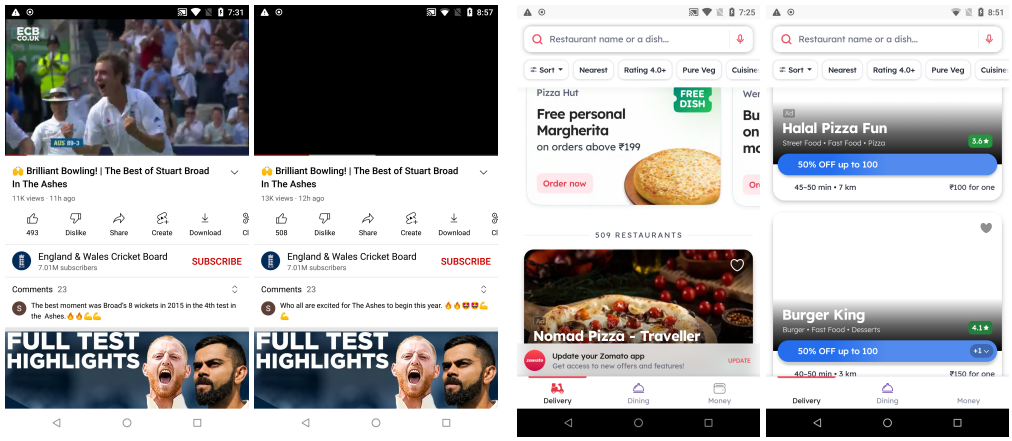
Additional Key Words and Phrases: mobile system, power consumption, development, OLED displays

1 INTRODUCTION

The quality of smartphone hardware, such as the compute capability of processors and size of display screen, has improved significantly over the last decade. However, the capacity of phone batteries has not kept pace with these improvements. Thus, smartphones are still energy-constrained. Numerous studies have demonstrated that high energy consumption exerts the most pronounced negative impact on user experience quality [16]. Moreover, reports indicate that over 90% of smartphone users experience low-battery anxiety [23, 42]. Manufacturing and disposal of used lithium-ion batteries also come at a significant environmental cost [29]. Hence, it becomes imperative to explore additional avenues for conserving battery energy.

A significant proportion of the power consumption comes from smartphone displays. A number of prior studies have proposed changing the content displayed on smartphone screens. For example, [12] uses a dark mode that disables colors from the screen. Focus [37] 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 [13, 35, 36]. As discussed by [12], the power savings obtained using these techniques is relatively moderate (< 10% of the total power). This is because these techniques target reducing the OLED or display screen power, which only forms a relatively small portion of the total power in modern

*These authors contributed equally to this work.



(a) YouTube app (Video Streaming) (b) Zomato App (Shopping)

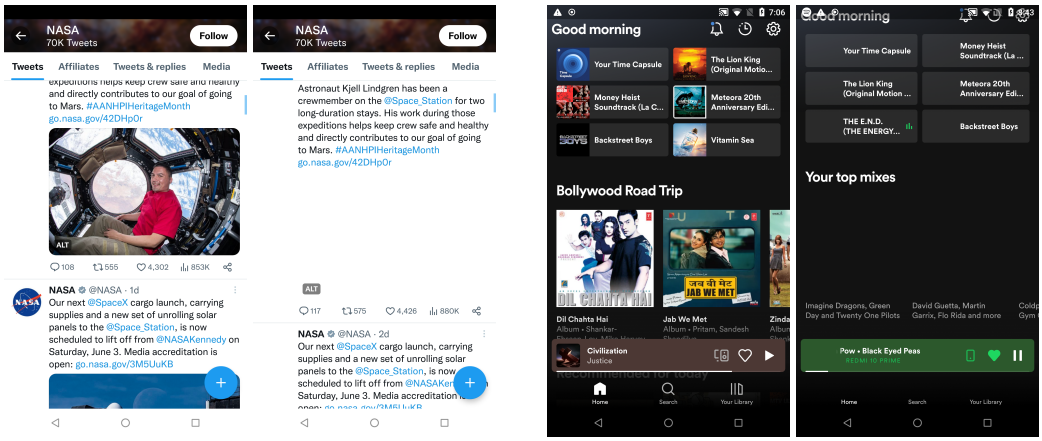
Fig. 1. FlexDisplay with YouTube and Zomato apps. Note that the user comments and other video thumbnails (at the bottom of the screen) can be seen as usual, but the video stream and images are disabled.

smartphone architecture. The limited power saving is due to the fact that these techniques cannot reduce the computation involved in rendering the content, as the computation takes up the majority of phone power consumption.

Our approach towards energy savings is motivated by a few observations. Firstly, today's applications or operating systems cannot selectively disable specific portions of the user interface (UI), even when certain UI elements are unnecessary or could be hidden in certain scenarios. For instance, in video conferencing applications like Skype or Zoom, a user does not have the option to stop displaying individual or all attendee video windows. This lack of flexibility becomes particularly evident in situations such as online courses, where most attendees' or students' videos are unnecessary and potentially distracting. Secondly, current smartphone systems and apps not only lack flexibility in display options, but they are also not optimally designed for power-saving considerations. Significant power is consumed unnecessarily. For instance, while dimming the screen brightness to zero or using a black screen can save some power, this approach falls short of its full potential because the UI content continues to render in the background. Although a user can barely see the UI in this case, the phone still consumes CPU and GPU computing resources at the same power level as before. Therefore, designing a system that integrates flexibility in display options along with an optimized rendering strategy is crucial, as it delivers the benefit of power savings with minimal disruption to the user experience.

In this paper, we introduce a framework called FlexDisplay, which puts these insights together, achieves selectively disabling UI elements to optimize smartphone power consumption, with minimal disruption to user experience. Figure 1 and Figure 2 illustrate how FlexDisplay disables UI elements, contrasting them with the original views of the original app. To achieve this goal, FlexDisplay provides users with a flexible display option for easy customization, allowing users to prevent unwanted or unnecessary UI elements according to their preferences. This customization significantly enhances power savings.

There are several challenges to achieving the goal in FlexDisplay. The first challenge is about how to achieve the selective rendering design. A crucial requirement for selective rendering is ensuring that disabled components do not necessitate processing by the smartphone's system-on-chip (SoC)



(a) Twitter App (Social Network) (b) Spotify App (Podcast)

Fig. 2. Examples of screenshots of original apps (left) and FlexDisplay-mode version (right), from the categories of Social Network and Podcast.

CPU. Another noteworthy aspect of FlexDisplay is its capability to offer end-users options to disable specific parts of the content based on their preferences. For instance, users have the flexibility to not display the video playback window while keeping the audio active and browsing comments on the video, or they can opt to disable avatar pictures in a chat window without even letting the other participants know about it. While a few current apps, such as Google Meet recently allow such disabling of videos [30], FlexDisplay provides this facility across apps, relieving the developers from specifically implementing this feature. This customization empowers users to tailor their content viewing experience, conserving power by displaying only the essential information relevant to their needs. To address this challenge, FlexDisplay enhances the Android View Service in the Android Framework layer by introducing a selective display module, which enables flexible portion display to cut off unnecessary UI portions, according to the preferences of users.

The second challenge handled by FlexDisplay is about how to maintain usability when disabling UI elements. Disabling a segment of the UI introduces potential usability issues. For example, users might unintentionally tap on the disabled (invisible) UI element, such as an invisible image button, triggering new windows or actions that can cause confusion and disrupt the overall user experience. Regarding 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.

The third challenge involves generalizing flexible display technique to scale up across more apps. Manually updating each app's source code for flexible display UI design is time-consuming and resource-intensive, requiring substantial development and release efforts. To tackle this challenge and extend the power-saving approach to a broader range of apps, FlexDisplay introduces the concept of universal naming aliases for UI elements. By creating universal naming aliases for UI elements, FlexDisplay simplifies the process for users to specify which UI elements they want to disable or modify. FlexDisplay uses a static analysis tool to decompile the app and get the layout

and resources files. It then uses a heuristic to find out the appropriate class names for UI elements from the layout and resources files, present within the packaged app. Once the configuration file is parsed by FlexDisplay, it provides a user-level API that the operating system can utilize to obtain the user's preferences. This user-level API serves as a means for the operating system to understand and implement the user's power-saving preferences effectively.

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 both the power savings using FlexDisplay on two different smartphones and their usability via user studies. Our evaluation includes apps across a wide range of categories, namely, 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 Wall Street Journal (news category) and Skype (video conferencing category) respectively. We further compare with darkening of display as used in prior studies to show that our power savings are much larger. This shows that FlexDisplay drastically saves power and prolongs battery life.

We further conduct an IRB-approved user study to check if users would be willing to use FlexDisplay. We invited a total of 20 participants and asked them to complete certain actions for each app e.g., create a new playlist of 5 songs on Spotify with and without FlexDisplay. We measure the completion time of the task with and without FlexDisplay. In addition, the users were asked to rate all the 15 apps in terms of usability from 1 to 5, with 1 denoting least usable and 5 denoting most usable. We observe that the users do not have any major difference in completing the task with and without FlexDisplay. We get a median rating of 3.8 for 15 apps. Furthermore, users were also asked to rate the overall idea of using FlexDisplay, and gave a median rating of 3.9 across all 15 apps, indicating that most of them liked the overall experience.

We summarize our contributions as follows:

(1) **Identification of Use Cases:** We first identify a number of use cases where users can utilize apps while disabling a part of the display. This includes apps across a wide variety of categories.
(2) **Design of FlexDisplay Framework:** We design a framework FlexDisplay, which disables the rendering of some user interface component to save power. FlexDisplay achieves this by introducing additional features and modifications at the View system within the Android framework. FlexDisplay further ensures that disabling this user interface component also automatically ignores the taps on the touchscreen of the corresponding area, by making changes in the relevant driver within the kernel. We have open-sourced each component of the source code of FlexDisplay to enable its utilization by the research community[1].
(3) **Quantification of Power Savings:** We extensively evaluate the power savings of FlexDisplay on Google Nexus 6 and OnePlus 3. We have released the power measurement data[2]. We show that FlexDisplay can save a median of 14.59% of the total power consumption, across a wide variety of apps. This indicates that FlexDisplay both saves significant amount of power while also being usable in practice.
(4) **User Studies to Quantify Experience:** We performed a user study on a total of 20 users, and show that they do not have major differences (median difference of 1.3%) in time spent in performing a given task. We also ask users to quantify their experience on a score of 1-5 (5 being the best), and received a median usability score of 3.9/5.

[1]Source code available at https://drive.google.com/drive/folders/1aKPWtW4i5zP2gP3Kycqw9LkhPhaSfv8o?usp=sharing.
[2]https://github.com/sahil20021008/flex-display-data

A preliminary version of this work has appeared at [14]. This version enhances the work by: (i) reducing manual effort in user interface annotation, (ii) scaling up the study to a second smartphone, (iii) quantifying the overhead of FlexDisplay in terms of power consumed, and (iv) performing user studies on a total of 20 users to show the effect of FlexDisplay on quality of experience, and (v) adding a significantly enhanced discussion of related works, (vi) comparing our energy savings with a baseline technique, and (vii) microbenchmarking the power implications of disabling touch sensors.

2 USE CASES

The key observation motivating FlexDisplay is that apps today display a significant amount of information that users do not need. For example, a podcast app like Spotify shows the images of the albums. Similarly, news, sports, and social networking apps show images related to events. Disabling the rendering of such content would save significant power without disrupting user experience.

In addition, FlexDisplay can also be used for the following possible use cases:

(1) **Listening to audio content using video streaming apps:** A large number of users frequently utilize video streaming apps like YouTube for listening to podcasts or music [28], where the video content being displayed may be redundant, for example, it could consist of a music video or simply natural pictures accompanying the audio content. Furthermore, the demand for audio content consumers is rapidly increasing [3]. However, these apps lack the option for users to disable the video portion, leaving only the audio content playing. This leads to unnecessary power consumption.
While some video streaming apps do offer the feature to play audio in the background, this often requires a subscription, such as YouTube Premium. However, even with a subscription, the app does not allow users to disable only the video content within the app, but permits the video to run in the background without displaying any content of the app. This limitation restricts users from accessing other preferred content such as comments and browse other recommended videos when listening to the audio in parallel.

(2) **Selectively disabling attendee's video window on video conferencing apps:** In video conferencing applications such as Skype or Zoom, users lack the option to stop displaying individual or all attendee video windows. This limitation can be particularly distracting, for example, in online courses where most attendees are students, and their video feeds may not contribute directly to the course content. Additionally, the presence of numerous video windows increases power consumption. Therefore, a flexible option to disable any attendee's video window is essential for maintaining concentration and achieving power savings. Recently Google Meet has introduced a feature to allow a user *to not watch* the video of each participant separately [30]. Such a feature highlights the need to for turning off video of other participants without informing them. Note that having an audio call for such use cases is not suitable as some participants might be interested in watching the video while others might not be.

(3) **Repeated purchase of online products on ecommerce platforms:** Surveys indicate that approximately 28% of total customer orders on e-commerce platforms, including food delivery and shopping apps, are repeat orders [38]. For these repeat orders, customers do not necessarily need to view large images of the products, which consume significant power to display, in order to appeal to them.

(4) **Recording long videos:** Users may often use smartphones as video cameras for extended recording sessions, such as capturing lectures, concerts, or other events. During these scenarios, users often keep the phone still for prolonged periods. However, the continuous display of

the recording preview on the screen is unnecessary, as the preview remains consistent and there is no need to constantly monitor it. This perpetually active display of the recording preview represents an inefficient energy design and can contribute to unnecessary power consumption. However, all camera apps currently lack the functionality to disable this display, thereby resulting in wasted energy.

3 BACKGROUND

In this section, we initially delve into the functionality of Android apps regarding their interaction with different components within the Android architecture. Subsequently, we explore the power consumption attributed to the display. Following that, we introduce our rationale and motivation for the design of FlexDisplay.

3.1 Working of Android Apps

The Android architecture consists of a total of four components – the Hardware Layer, the Linux kernel, the Application Framework consisting of both native and Java libraries, and the Apps (shown in Figure 3). The Linux kernel handles low-level communication with the hardware, as it contains individual drivers. The Framework has a hardware abstraction sublayer to provide a generic interface to the underlying hardware, so that the library functions do not require customization depending on the underlying smartphone model. It also has a number of common frameworks such as View Manager, Notification Manager and Location Manager to allow easier access to the common requirements of apps. At the highest layer lies the apps themselves, typically programmed using either Java or Kotlin.

Thus, the content displayed on smartphones is controlled by four different components of Android. The Apps decide the actual display content in terms of individual user interface elements, such as buttons or textboxes, in the form of a list. Different types of UI elements, such as buttons, text views, images, layouts, etc, have different classes associated with them. When a layout is rendered, the Android framework creates an instance of each class used in the layout and sets its properties and attributes based on the XML markup. Android framework uses the rendering pipeline to convert the layout into a series of pixels displayed on the screen.

The Android View system defines the components which are part of the Application Framework. Within the framework, android stores the information about what it needs to render on the screen in a list called as DisplayList. This DisplayList contains the location and the hash values of the pixels that need to be drawn. Decisions about which component to use and their sizes are taken by the apps on the CPU. The responsibility of rendering these components, i.e. converting them into individual pixels is performed by a library of the Hardware Abstraction Layer, typically on the GPU of the system-on-chip. Once decided at the pixel level, the Linux kernel handles the actual sending of interrupts to maintain a proper refresh rate of the display. A recent study has shown that this View system is largely similar across a wide variety of smartphones [25].

3.2 Power Consumption of Displays

Since smartphones are energy-constrained, a number of studies have measured the power consumption of different components [33]. These studies showed that one-third of the total energy of smartphones is consumed by the display. Modern smartphones typically use Organic light-emitting diodes (OLED) for display, which do not require any backlight. OLED displays, therefore, have the advantage of reducing energy consumption by customizing the display content. Furthermore, OLED displays also enable adjustable dimming of the LED's to further conserve energy.

A number of prior works have, therefore, proposed darkening the parts of the screen or reducing brightness that are less relevant to users [12, 32, 35]. For example, the work Chameleon [13] reported

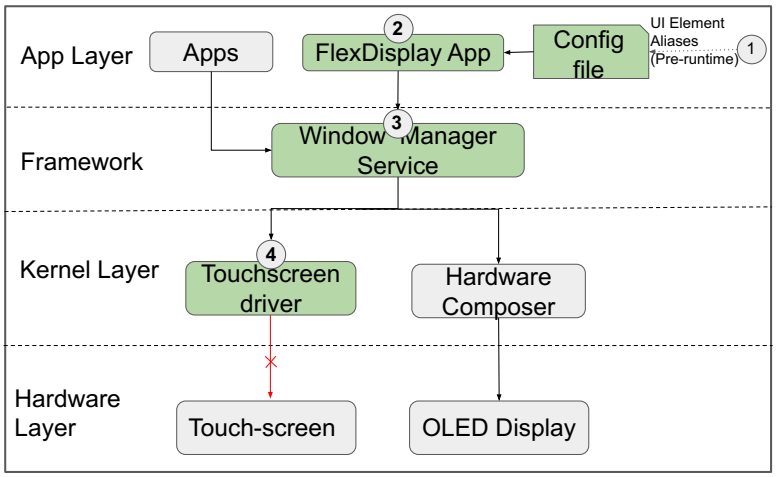


Fig. 3. Architecture of FlexDisplay in the context of the entire Android software stack. The highlighted blocks in green represent the components that FlexDisplay has either added or modified.

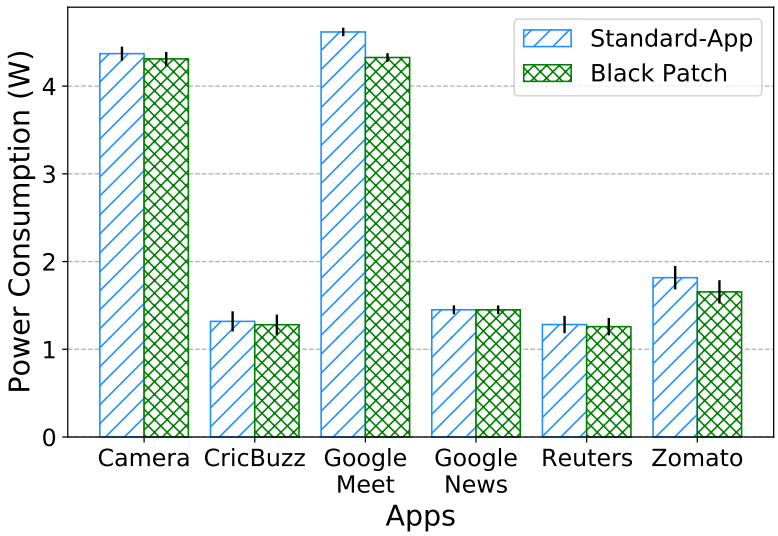


Fig. 4. Power savings obtained using only black patch over the entire screen.

(in 2011) 60% of the total power consumption being incurred due to the OLED display. However, recent years have seen processors with more compute capability gradually being incorporated into smartphones, thus increasing the power consumption of processing. Thus, the amount of smartphone energy conserved by screen darkening has progressively reduced over the years, and is less than 10% in modern smartphones assuming 50% brightness [12]. This suggests the need for an alternative approach to conserve energy.

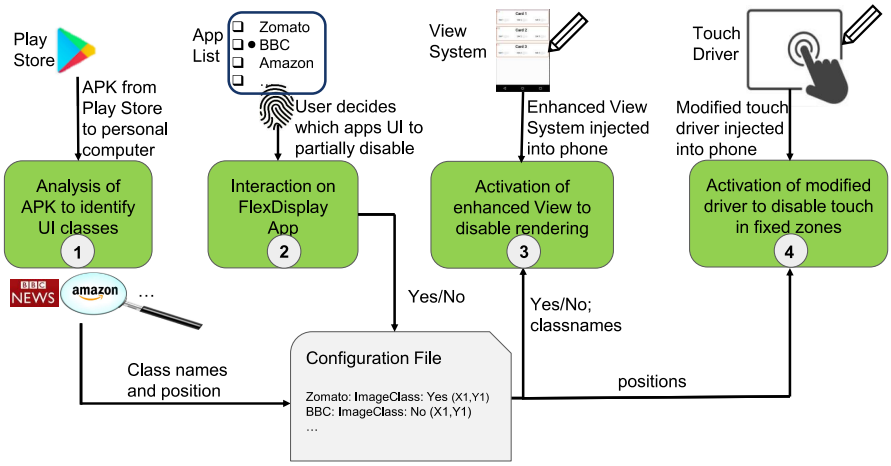


Fig. 5. The four components of FlexDisplay. The components shaded blue are done as a preprocessing step, while the ones shaded yellow are run by the Android framework at runtime.

3.3 Motivation

We identify the scope of minimizing the display power as follows. OLED displays are known to consume "minimal" power when their pixel brightness is set to 0 [12, 13]. We, therefore, set all the pixels to 0 by creating an overlay app that displays a black patch on top of the screen. We then measure the power consumption both using the original app (without any user action) and while applying the black patch using a Monsoon Power Monitor (further details are given in §V). Figure 4 shows the amount of power saved using this technique on a few representative apps. We note that the power saved is always lower than 9%, with the median power saved being close to 4%.

Since the black patch overlay is a service running over any other activity on the screen, we further follow an alternative technique of blackening the screen. In Android, each frame on the screen consists of multiple layers composed together for example navigation bar, status bar and on screen activity. All these layers are sent to Hardware Composer (HWC) to compose them together. We altered the HWC to crop the activity layer. As a result, the cropped part is rendered black. This approach adds little to no added overhead, in contrast to the black patch overlay. However, the power consumption numbers after equal percentage of screen blackened using the black patch overlay and HWC cropping are almost equal.

This confirms the observation by recent prior works such as [12] that the power consumption from smartphone's displays is a small proportion of the total power.

4 DESIGN GOALS AND OVERVIEW OF FLEXDISPLAY

4.1 Goals of FlexDisplay

The key goal of FlexDisplay is to save power while minimizing disruption to the user's ongoing activities within apps. Achieving power Saving requires us to identify the power consumption sources within the rendering process. We identify three major sources of power consumption where optimizations are possible:

(1) **Display of pixels by OLED:** The power consumption of OLED display depends on the actual color and brightness level of each individual pixel. If the pixel is completely darkened, then no power is consumed, thus saving power. Thus, a number of prior works have utilized darkening [12], brightness dimming [36] and variable refresh rates [27] to reduce this component of the power consumption.
(2) **Cost of UI re-construction by CPU:** Any changes in the UI, such as creation of a button or changing of the screen, requires CPU computation. The CPU generates a list of UI elements along with their semantic information such as coordinates, size and textual information by calling an update process that is facilitated by the Framework Layer. This imposes a substantial overhead due to the computation involved. However, none of the prior works using changes to the displayed content have reduced this component of the power consumption.
(3) **Cost of taps on the touchscreen:** Touchscreens of smartphones are typically capacitative, and they do not consume power until the user actually taps them. However, users may mistap the portion of the display where the UI is disabled, thus costing additional power.

FlexDisplay aims to conserve battery by reducing the power consumption of all the three above components. We now explain the design choices to enable such power savings.

4.2 Possible Design Choices

Unlike previous systems, FlexDisplay reduces the power consumption of all three components. There are multiple possibilities for achieving it. First, we could change each individual app, at the bytecode level to remove some UI components while displaying the images/videos. The key drawback of this approach is that all the apps need to be statically analyzed and the changes incorporated. A second possibility is to disable parts of the display in the display driver present in the kernel layer. However, this would make FlexDisplay less portable or scalable as the approach would need modification of the driver of each smartphone. Furthermore, it also loses the semantic information of the UI elements, making it difficult to identify what exactly to disable.

Thus, FlexDisplay takes a third possibility, upgrading changes at the Android Framework Layer as shown in Figure 3. The broad technique of FlexDisplay is to disable the generation of selected (by user) UI elements while generating them. UI generation takes up significant CPU power, which we will demonstrate in Section 6. To reduce the power, FlexDisplay provides an upgraded View System with the application framework and provides the user the option to decide which parts of UI to disable, such as images or videos UI parts that consume power the most. This has the advantage that (i) the UI elements retain their semantic information in the Android Framework Layer, so it is easier to disable them individually, and (ii) disabling the entire UI element also makes the display more intuitive.

However, partially disabling UI brings another challenge: some of the disabled UI may contain clickable UI, such as buttons, scroll lists, etc, which would pop-up unexpected windows or contents and cause confusion if users mis-tap the location where the UI becomes invisible or disabled. To avoid such confusion due to mis-tapping, FlexDisplay prevents inadvertent actions by disabling the screen-touching function of the invisible UI. This implies that even when users inadvertently touch the screen region of the invisible UI, nothing will be triggered. This design is challenging because the touching event is normally enabled or disabled in the lower layer, i.e., Hardware Abstraction Layer (HAL) which loses all semantics information. This implies that HAL is unaware of the UI element that needs to be disabled to touch input. To address this problem, FlexDisplay incorporates semantics information from Framework layer. It extracts the screen coordinates of UI from Framework Layer, and passes it to a configuration file (more details in Implementation

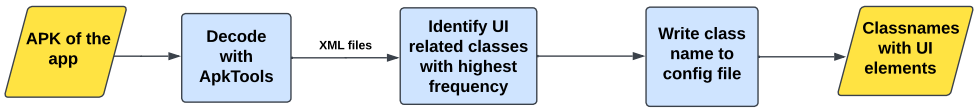


Fig. 6. Our technique of generating the config file using static analysis of APK code.

Section)[3]. Overall, this design approach allows FlexDisplay to provide finer control over the partial disabling of UI display and touch input, thereby improving power consumption and enhancing the user experience as well as mitigating confusion caused by inadvertent actions.

4.3 Steps Involved in Designing FlexDisplay

FlexDisplay is designed in four major steps as shown in Figure 5. ❶ 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. ❷ 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. ❸ 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. ❹ 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, as shown in Figure 3. 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. And 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.

5 IMPLEMENTATION OF FLEXDISPLAY

In this section, we discuss in detail the implementation of each of the four steps of FlexDisplay. Our implementation strategy is motivated by the fact that we need to identify the right user interface elements, while also ensuring that the amount of human effort required in identifying them is minimal. Since each app is developed by its own set of developers, it is non-trivial to identify the right user interface elements. We explain how FlexDisplay's implementation handles such challenges.

5.1 Build Naming Alias of UI Elements

The first challenge of FlexDisplay is how to establish an intuitive method for users to effortlessly specify the UI elements they wish to disable. Since users may lack familiarity with app design and don't know the name corresponding to each UI element, it becomes challenging for them to effectively specify UI preferences to the operating system. Breaking this information barrier

[3]Note that Focus [37] also took a similar approach of implementation; but its goal was to implement dimming of portions of the screen to save OLED power. This meant that Focus did not need to explicitly identify any UI element. However, this implies that incurring CPU power due to the rendering of the UI element cannot be avoided.

```
1 <ZRoundedImageView android:id="@id/bg_image" ... />
2 <ZRoundedImageView android:id="@id/full_image" ... />
3 <ZIconFontTextView android:textSize="@dimen/height75"
      android:textColor="@color/sushi_white"
      android:layout_gravity="center" android:id="@id/popup_icon"
      ... />
4 <ZRoundedImageView android:layout_gravity="center"
      android:id="@id/alert_image" ... "/>
```

Fig. 7. A fragment of an UI layout file from an Android APK. Note that all the UI elements corresponding to images have the value with substring “image" for the attribute “android:id", whereas this is not so for the other UI elements. We have omitted additional attributes in the interest of simplicity.

between users and applications poses a considerable difficulty, especially for developing a universally applicable solution. Although an approach of manual annotation of each UI element is possible using the tools provided by Android Studio, this requires considerable manual effort and stops functioning with each function upgrade.

We, therefore, utilize static analysis of the Android Application Package (APK) of each app, downloaded from the app stores. Figure 6 shows the steps for this process. We utilize *apktool* to decode and decompile the APK into human-readable form. Android allows the UI to be defined in the form of XML files. These decoded APKs have the XML class layout files with classes defined for each of the UI elements, along with other attributes. Each class has an identifier associated (highlighted in Figure 7) to enable the developers to access it from the source code. Android Studio by default adds an “image" appended to each such identifier. We hypothesize that most developers retain a similar substring as it is intuitive to use it. This is especially true because such names are typically auto-generated using the tools that Android Studio provides, as opposed to the developers creating new names. We confirm our hypothesis by verifying that all the 15 apps we tested have the substring “Image" for the UI elements corresponding to images or videos. This enables us to identify the potential content to be disabled.

However, we avoid disabling all such images or videos, as many of them may represent important UI components, such as icons or image with links used for user interaction. Thus, our next attempt is to identify the classes that appear most frequently, as we note that app icons and other similar images are rarely used. Moreover, the most frequently used image/video classes also consume the most area, and are therefore, likely to consume most power during rendering. We, therefore, try in each app the three most frequently present classes in each activity. We find that in 11 out of 15 apps, the image/video we plan to disable corresponds to the most frequent appearing class. In the remaining 4 apps, they correspond to either the second or third most frequent case. We acknowledge that this requires a small amount of effort to enable FlexDisplay for each app. However, it does not require access to the source code and requires only minimal amount of effort ($<$ 10 minutes per app). Furthermore, since an app is unlikely to change its identifiers on version upgrades, it does not require re-identification of the classes. Moreover, this technique generalizes across all activities of the app, making it easier to scale to many apps. This technique is also not affected by the apps whose interface changes frequently. Our primary focus is on user interface elements that are retained on the screen, as they consume power continuously. However, this technique works irrespective of how quickly the user interface changes, as the user interface elements are identified

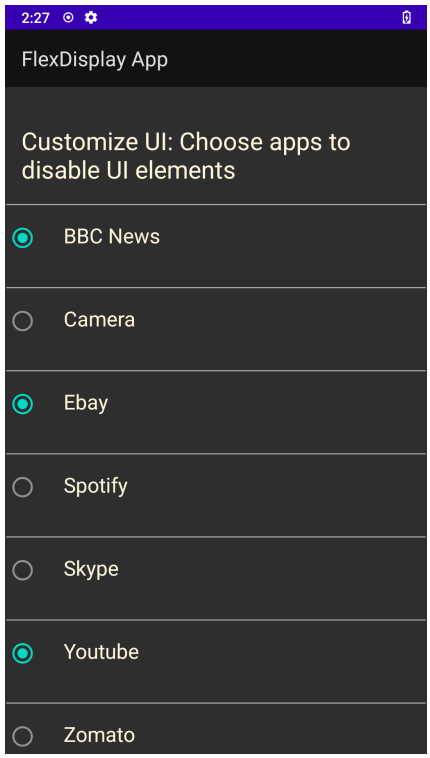


Fig. 8. Screenshot of FlexDisplay's app interface.

a priori. While not observed in the 15 apps we have tried, we recognize that this technique might occasionally miss the right user interface elements (more detailed discussion in §9). The classes so identified are stored in a config file corresponding to each app. This config file is then transferred to the smartphone for use by FlexDisplay, either over the network or using a SD card.

5.2 Customize UI for Partial Display

This is the only step while utilizing FlexDisplay that requires human-in-the-loop involvement from users. It is not feasible for users to directly modify the class methods, as that requires significant programming background and context.

We, therefore, have developed a FlexDisplay app shown in Figure 8 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. As users modify the app settings, the FlexDisplay app updates the configuration file by adding a "yes" or "no" designation alongside each class name.

Eventually, FlexDisplay stores the updated configuration file 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,

```
class DisableView {
    String[] classesDisabled;
    boolean checkConfig(String ui) {
         if (ui is in self.classesDisabled)
             return true;
          else
             return false;
      }
  }
class View {
    invalidate(String ui) {
      if (!DisableView.checkconfig(ui)) { /*Run usual code*/ }
    }
    draw(String ui) {
      if (!DisableView.checkconfig(ui)) { /*Run usual code*/ }
    }
    updateDisplayListIfDirty(String ui) {
    if (!DisableView.checkconfig(ui))
         { /*Run usual code*/ }
    }
 }
```

Fig. 9. Pseudocode showing how FlexDisplay utilizes the DisableView class to check the configuration file and disable or allow rendering of UI components. The classesDisabled string is separately read every second from the external memory card by a different thread and updated.

the system creates a bridge that facilitates the transfer of user configurations across different apps. This approach allows seamless sharing and synchronization of user preferences while adhering to Android's security considerations.

5.3 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 actual technique used to disable rendering is shown in detail in Figure 9. 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 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

Algorithm 1 Ignoring interrupts in the Synaptics touch kernel module

```
1: procedure SYNAPTIC TOUCH REPORT(Fhandler)
2:     data ← Fhandler->data
3:     x_coordinate = data− > x
4:     y_coordinate = data− > y
5:     if x_coordinate and y_coordinate in disabled area then
6:         return 0
7:     else
8:         Report the touch
```

to all UI elements across all apps. Hence, the UI render can be automatically disabled once the user completes the UI customization.

5.4 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.

We first load the config file as a special file within the privileged file system group. This allows the kernel to obtain the region information where the touch input needs to be disabled. A pseudocode of the implementation is shown in Algorithm 1. Whenever an interrupt is triggered due to the touch, our modified kernel module checks whether it falls within the disabled region (Line 5). Note that we only allow rectangular areas to be disabled, so the check consists of just a few comparisons of the coordinates. If the check shows that the area is disabled, we then ignore the interrupt (Line 6) and return without reporting the touch coordinates. Otherwise, we let the usual reporting of touch (Line 8).

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 and OnePlus 3 smartphones, we utilized the Synaptics touch module [2]. 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.

6 EVALUATION

We will now explore the power savings achieved through FlexDisplay. We conducted power measurements on two smartphones, namely the Google Nexus 6 and OnePlus 3. Note that the power measurement requires connecting it to the Monsoon Power Monitor, which requires tear-down of a smartphone and soldering wires to attach the smartphone to the Monsoon power monitor [4].

[4]Those experiments involving power profilers that utilize power models are not compatible with FlexDisplay, as these models cannot account for alterations made in its stack. Thus, we need a hardware power monitor to measure the power consumption.

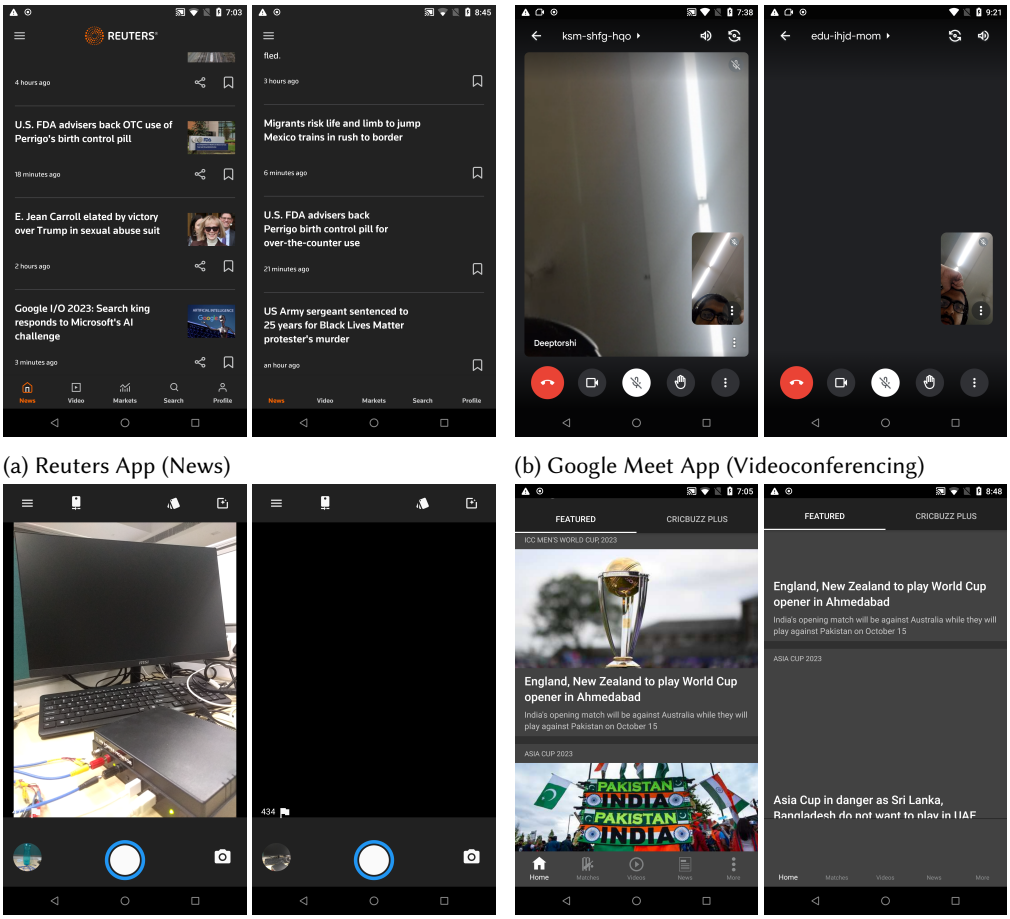
Table 1. A list of apps along with their categories and the components of UI's disabled by FlexDisplay.

App Category	App Name	Disabled Part(s)
News	Reuters	Images in articles
	Wall Street Journal	
	Google News	
	InShorts	
Shopping	Zomato	Preview images of products
	Ebay	
Video Conferencing	Google Meet	Video of other participants
	Skype	
Video Streaming	YouTube	Parts of the screen that streams or runs the video
	HotStar	
Camera	Camera	Video preview
	OpenCamera	
Sport	Cricbuzz	Images of articles & videos of discussions
Social Network	Twitter	Profile photo & Images shared
Podcast	Spotify	Album cover for songs

6.1 Evaluation Setup

We selected a total of 15 apps across 7 different categories – news, shopping, video conferencing, video streaming, sport, social networking and podcast. Each of the selected apps have over 1 million downloads as reported by Google Play Store. Table 1 presents the app, along with categories. We use a Monsoon Power Monitor [1] to measure the power consumption in a room at ambient temperature within a period of one month. We disabled all background apps before starting the measurement. For each app, we measure phone power consumption over a 2-minute duration, repeating the process for alteast five times. Subsequently, we compute the average and standard deviation values. We further disable image and video UI elements using FlexDisplay and conduct a new set of measurements for five iterations, each lasting 2 minutes. For applications where users explore content through scrolling, such as Ebay, we employ a script for automated scrolling. This script operates in a loop, navigating up and down, and employing swipe gestures to traverse pixels.
Baseline Techniques: As a baseline, we capture measurements with a black overlay covering 100% of the screen[5]. We do not compare power savings with Focus [37], as the black overlay technique delivers greater power savings than the partial dimming achieved by Focus. We do not compare with orthogonal techniques that scale frequencies intelligently using DVFS [8], or utilize dynamic frame rates [17] as they are not directly comparable with our approach, and could be used alongside FlexDisplay. Note that the power-saving modes frequently employed by smartphones when operating under low battery conditions incorporate a suite of energy optimization strategies. These typically include Dynamic Voltage and Frequency Scaling (DVFS), which adjusts the processor's operating voltage and frequency to reduce power consumption; lowering the display refresh rate to minimize GPU usage; and aggressive throttling or suspension of background network activity, including reductions in data transfer frequency and throughput. As a result, these simultaneous optimizations

[5]Although none of the techniques darken the entire screen, we adopt this approach to maximize power savings on OLED. This is because black screen conserves the largest amount of power compared to transformation of color in any form.



(a) Reuters App (News)

(b) Google Meet App (Videoconferencing)

(c) Camera App (Camera)

(d) Cricbuzz App (Sports)

Fig. 10. Screenshots of one app in News, Videoconferencing, Camera, and Sports categories, original versions(left) vs FlexDisplay-mode versions(right).

significantly change the phone's performance characteristics. Thus, a direct one-to-one comparison with our proposed approach becomes impractical and potentially misleading.

A limited number of applications, including YouTube and Google Meet, offer the capability to continue functioning in the background, even when the user navigates away from the app interface. For these two applications, we include their background operation modes as additional baseline scenarios in our analysis as such background operation will not incur display power consumption. Although Spotify similarly supports background execution, we exclude it from such baseline comparisons. This is because Spotify primarily delivers audio content with relatively static or non-changing visual elements on the screen, such as album art or a static interface. Consequently, its display-related power consumption remains largely unaffected whether the app is running in the foreground or background.

Phone Models: All experiments were conducted on 1) Google Nexus 6 smartphone and 2) OnePlus 3 phone. The Nexus 6 phone features a display size of 5.96 inches, 97.9 cm2, with a resolution of

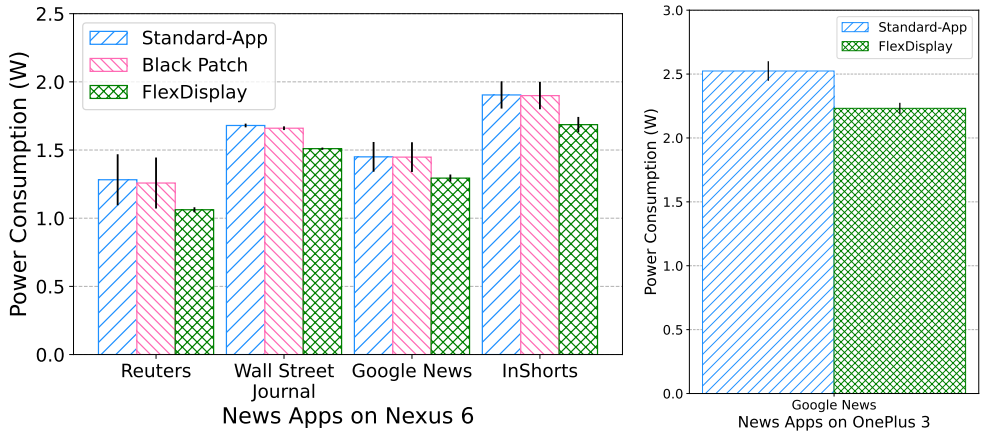


Fig. 11. Power savings obtained using on apps of news category using standard app & black patch over the entire screen, and using FlexDisplay for Google Nexus 6 and OnePlus 3.

1440 × 2560 pixels and a 16:9 aspect ratio. The device is equipped with a Qualcomm APQ8084 Snapdragon 805 (28 nm) chipset, featuring a Quad-core 2.7 GHz Krait 450 CPU and Adreno 420 GPU. The OnePlus 3 phone features a display size of 5.5 inches, 83.4 cm2, with a resolution of 1080 × 1920 pixels and 16:9 aspect ratio. The device is equipped with a Qualcomm MSM8996 Snapdragon 820 (14 nm) chipset, featuring a Quad-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo) CPU and Adreno 530 GPU.

6.2 Apps using FlexDisplay

Screenshot of Apps using FlexDisplay: We now compare the changes in the interface seen in the apps once we use FlexDisplay in Figures 1-10 and 2,. To save space, we show the interfaces only for one representative app in each category. We visually note that the changes in the Reuters, Spotify and Cricbuzz lead to removal of decorative images. It is also clear that Spotify and Cricbuzz have a higher opportunity of saving power using FlexDisplay because the images occupy larger space on the screen. We further note that Google Meet and the Camera apps disable a large portion of the content shown on the screen. However, even apps where relatively smaller amount of content is disabled, have at least close to 10% of power saved.

6.3 Power Savings by Selective Rendering of User Interface

We show the power consumption on each of the fifteen apps, based on category (Figures 11 – 16). We have released the raw values of power measurement data for both phones[6]. The raw data is available both in CD5 and CSV formats. The image of the readings is also available. Each measurement is for 2-minute duration. Each measurement shows time, average current drawn in amperes, and average power drawn in watts. For each phone model, we show the power savings for each individual app with and without optimization. The summarized observation of average power and current drawn and their standard deviation across 5 readings are shown in the file “observations.txt". In addition, we show the power measurement for black patch and disabled touch on the Nexus phone. Since each of the different category of apps have diverse levels of power consumption, we plot them separately. We discuss the power savings of each category of apps:

[6]https://github.com/sahil20021008/flex-display-data

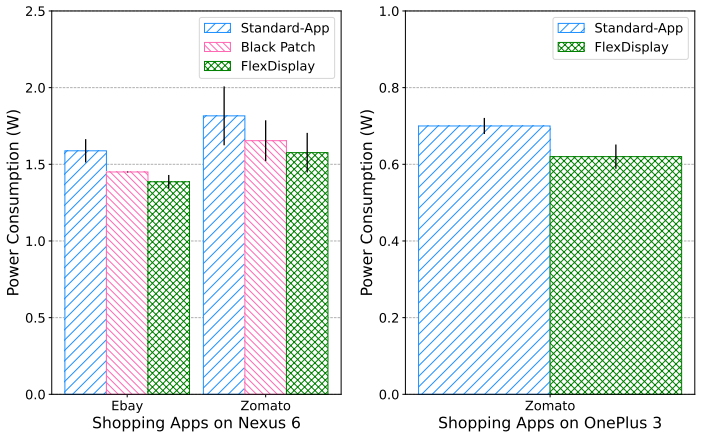


Fig. 12. Power savings obtained using on apps of shopping category using standard app & black patch over the entire screen, and using FlexDisplay for Google Nexus 6 and OnePlus 3.

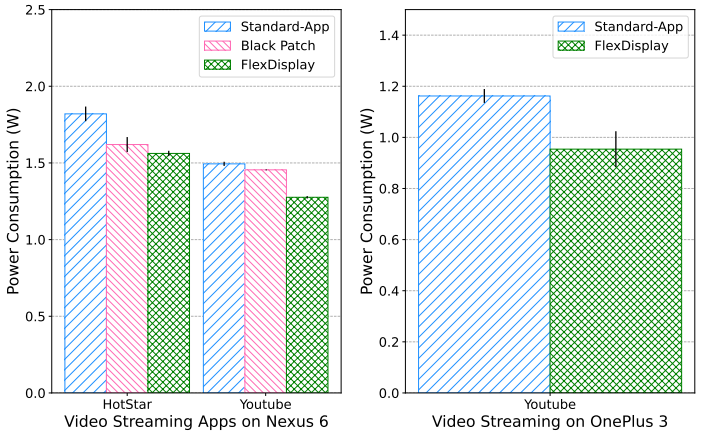


Fig. 13. Power savings obtained using on apps of video streaming category using standard app & black patch over the entire screen, and using FlexDisplay for Google Nexus 6 and OnePlus 3.

News Apps: We disable parts of the display that render images and videos (Figure 11). We observe that using FlexDisplay saves a power of 4.83%, 10.11%, 10.75% and 11.45% for Reuters, Wall Street Journal, Google News and InShorts respectively for Nexus 6. We observe that for OnePlus 3, the power saving is 11.57% for Google News. Such savings are similar to Nexus 6, possibly due to a similar screen size. We note that for all but one case, the power savings exceed 10%. The relatively smaller savings for Reuters can be explained by the fact that it has fewer number of images or videos in the content.

Shopping Apps: We disable the images of the products for the shopping apps (Figure 12). We note Ebay, a shopping app and Zomato, a food delivery app, show a power savings of 12.72% and 13.32% respectively for Nexus 6. A similar improvement of 11.42% is achieved for Zomato on OnePlus 3. In contrast, the black patch only saves 6.4% and 6.7% of power respectively each case.

Video Conferencing Apps: For video conferencing, we set up a video call with a single other participant and we disable only the video part of the other person. Note that video conferencing apps available today allow a user to switch off their own video, but not of the other participants.

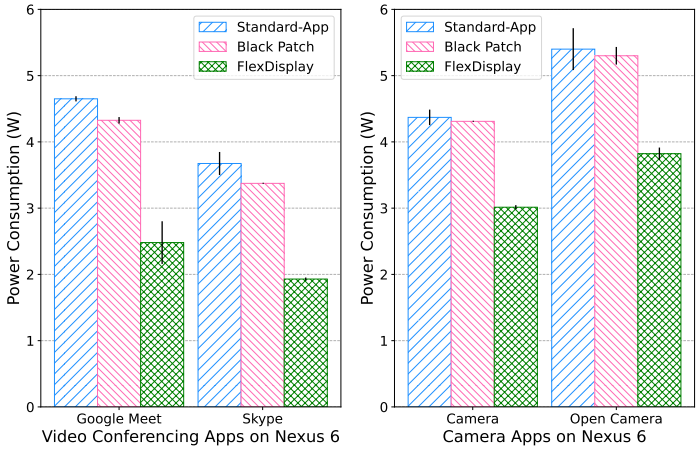


Fig. 14. Power savings obtained on video conferencing and camera apps using standard apps & black patch over the entire screen, and using FlexDisplay on Google Nexus 6.

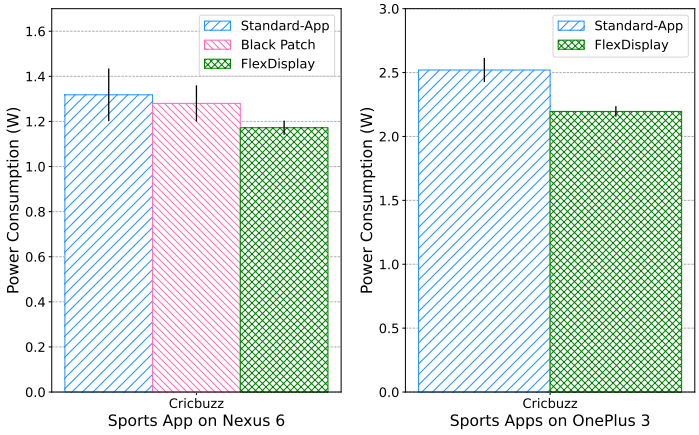


Fig. 15. Power savings obtained using on apps of sports category using standard app & black patch over the entire screen, and using FlexDisplay for Google Nexus 6 and OnePlus 3.

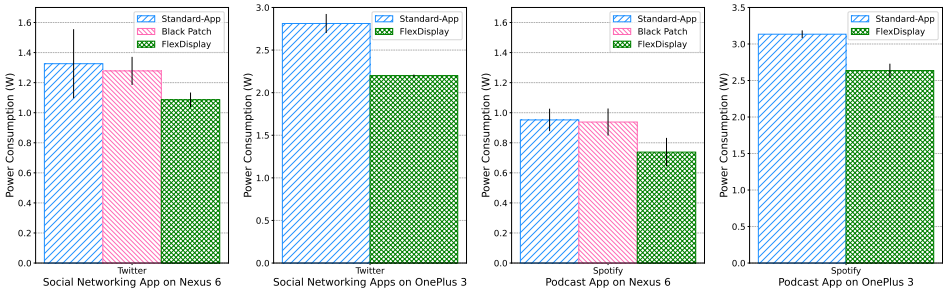


Fig. 16. Power savings obtained on social network and podcast categories using standard app & black patch over the entire screen, and using FlexDisplay for Google Nexus 6 and OnePlus 3.

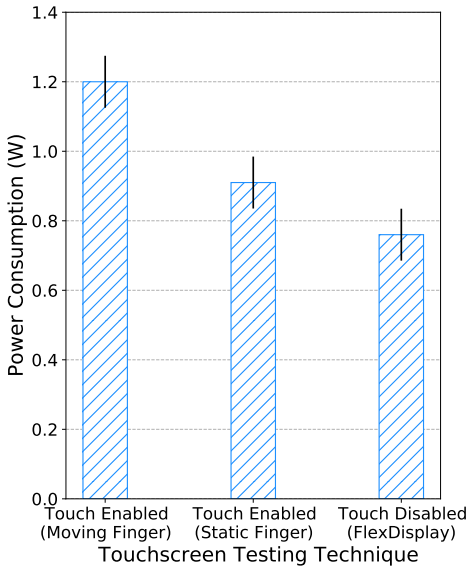


Fig. 17. Power savings obtained by FlexDisplay's by disabling the touchscreen.

We note that there is a significant amount of power savings as shown in Figure 14, with Google Meet and Skype reducing the power consumption by 46.67% and 47.74% respectively. On the other hand, disabling the video on Google Meet (using *not watch* a participant's video feature) saves a total of 51.57% power – a value comparable to the power saved using FlexDisplay. Note that this facility is not present on other apps like Skype. This shows that it is possible for batteries to last much longer using our technique when these apps are used.

Video Streaming Apps: For video streaming, we play a particular video on both YouTube and HotStar. Figure 13 show that they have a power saving of 14.59% and 14.18% respectively on Nexus 6. YouTube has a slightly higher power saving on OnePlus 3, it is 17.9%. Note that the power savings of video streaming apps is smaller than that of video conferencing, because these apps are assumed to be playing the video only over a part of the display. We further measure the playing of YouTube premium in background with the screen off[7], and find that it gives a power saving of 26.24% over conventional playing of YouTube. Note that using this mode does not allow users to read the comments or browse other video thumbnails.

Camera Apps: FlexDisplay disables the preview of camera apps, but allows the user to view the photo once it is clicked. For both Camera and Open Camera, a large part of the screen is disabled, hence both have large savings at 31.02% and 29.22% respectively as shown in Figure 14. This also explains why FlexDisplay saves a large amount of power on these apps.

Other Apps: We further study the power savings of three different apps – Cricbuzz from sports category, Twitter from social networking category and Spotify from podcast category shown in Figure 15 and Figure 16. These apps consume relatively lower power than the video or camera apps. FlexDisplay still saves 11.08%, 18.10%, and 22.48% of the total power, again showing a significant amount of savings on Nexus 6. On OnePlus 3, the power-saving numbers for Cricbuzz, Twitter, and Spotify are 12.85%, 21.76%, and 15.9%, respectively. Thus, we obtain somewhat similar improvement on both the phone models for these apps..

[7]The ordinary YouTube does not allow streaming of videos with the display off.

Summary of Observations: With an extended evaluation of 15 apps from 8 different categories, we observe that FlexDisplay provides over 10% power savings with a maximum of 47% for video conferencing apps. Video conferencing and Camera apps provide the most improvement, as in this case, a significant portion of the display is disabled. Interestingly, for most categories, different apps provide similar improvement, possibly due to the similarity of content in terms of the display portion occupied by media content. For example, eBay and Zomato provide a similar improvement of about 13%. Black patch only provides a power consumption similar to the original app. FlexDisplay always surpasses the power savings offered by the black patch. Further, we observe that both phone models provide a comparable performance improvement, possibly due to a similar screen size. However, we also note that absolute power measurement for both the phones varies significantly for the same app (for example, power consumption for the Zomato app for Nexus 6 and OnePlus 3 in Fig. 12). Such differences are possibly due to version differences of the same application between the phones.

6.4 Power Saving when Disabling Touchscreen

Disabling unnecessary touchscreen elements, such as stopping an image button from rendering and becoming invisible, gains two advantages: it not only mitigates user confusion caused by inadvertent mis-tapping but also contributes to power conservation. We conducted experiments to measure power saving when disabling the touchscreen. Since the power consumption depends on different user interaction patterns, we measure them in two scenarios: static tapping and dynamic swiping, to simulate users' inadvertently touching and swiping cases, respectively. In the first scenario, we used a script that automates tapping events for one minute; for the second scenario, we automated a routine swiping gesture on the screen for one minute. Finally, we measure the power consumption of FlexDisplay in both situations, though FlexDisplay's power consumption does not depend on the user interaction as the signals are ignored. We show the power consumption values in Figure 17. We observe that with touch being disabled the power savings are 36.67% and 16.48% with static and moving finger respectively.

These results indicate that disabling the screen touch for unnecessary portions of UI elements offers two benefits. First, it helps prevent users from accidental mis-tapping, thereby improving the overall user experience. It also results in significant power savings. By deactivating touch functionality in areas where it is not required, the device can conserve power resources by up to 36%.

6.5 Overhead of FlexDisplay

While FlexDisplay is designed with the goal of minimizing power consumption, it also introduces an additional layer of computation and associated overhead. To assess the extent of this overhead, we conducted measurements specifically focusing on the power impact introduced by FlexDisplay itself.

FlexDisplay has three sources of overhead – (i) overhead when a user interacts with FlexDisplay app when a user customizes and specifies a certain app's UI element to be disabled; (ii) System I/O overhead when FlexDisplay reads and writes the config file as per the user's preferences; and (iii) additional changes introduced in the enhanced View system. To quantify the overhead, we compared the power consumption between the original system without installation of FlexDisplay and the OS that equipped with FlexDisplay. From the result, it turns out that the difference in power between the hardcoded version and FlexDisplay is within 0.5 percent of a running app,[8]which is negligible.

[8]This small difference in power is difficult to quantify using direct power measurements.

App	Task Description
BBC News	Go to the "Explore" section and scroll down until you reach the section on "US Economy". Open the second article about the S&P index and read the closing price.
eBay	Search for five items, and add the first search result to the eBay cart. The five items were PS5, iPhone 14, Perfume, wallet, and sunglasses.
Open Camera	Take three photos of an item from different angles.
Spotify	Create a new playlist of 5 songs.
Skype	Join a meeting and then answer three questions. The questions were "What is your first name?", "What is your date of birth?" and "What is your branch?" in the first iteration of tasks, and "What is your last name?", "What is your place of Birth?" and "What is your year of graduation?" in the second iteration of tasks.
YouTube	Search Cricbuzz to find a specific video that explained a cricket match and play it. Then, listen to the video and report the score for the match when it was announced.
Zomato	Search for Domino's Pizza, add one Margarita Pizza and one Farmhouse Pizza to the cart.
Twitter	Navigate to your current profile and then post a Tweet on a specific topic.

Table 2. Tasks given to the participants in each category of apps.

Specifically, we dive deep and analyze each source of overhead by estimating their power consumption. We first note that interaction with FlexDisplay app requires a total of three screen touch gestures. Furthermore, Android defines a screen tapping event for a maximum of 100ms duration. Our power measurement shows that a screen tapping consumes a maximum of 0.9W power (according to Figure 17), leading to a total energy consumption of 0.27J of energy when interacting with FlexDisplay. We note that most of our chosen apps consumes at least 1W power without any user interaction (Figure 11-Figure 16), thus consuming 60J of energy per minute. Therefore, this overhead forms $0.27/60 = 0.45\%$ of the energy consumed by FlexDisplay app.

We also quantify the overhead of system I/O. To do so, we adjust FlexDisplay's write frequency to the local configuration into 4.6 writes per second, i.e., 550 writes for 2 minutes. We then compare the power consumption between the configuration updating scenario previously discussed and an idle mode (when FlexDisplay isn't updating the configuration). The result showed an increase in power consumption of 0.294W for 550 writes or 0.5mW per write. Note that we only utilize writes, because writes consume more power than reads on SD cards [15]. Since FlexDisplay initiates less than one input/output operation per second, thus, the power overhead is less than 0.05% of the app.

Finally, we note that our enhanced Android View System introduces no heavy run-time computation. Thus, there is no additional overhead introduced to disable the content. Furthermore, it is only used when an app is in the foreground. In total, the overhead comes to be $< 0.5\%$ running an app without FlexDisplay.

7 USER STUDIES

In this section, we introduce the user studies and show the usability and user feedback on their experience when using FlexDisplay.

We conducted an IRB-approved user study of the apps with the display being disabled with FlexDisplay. Since FlexDisplay has a simple and intuitive interface, we did not specifically ask the

participants about their experience when using it. During the study, users also showed no barriers to using the FlexDisplay app. The user study used two identical smartphones. The first smartphone runs the original version of apps, whereas the second uses FlexDisplay to disable image and video portions of apps. Note that both smartphones have the same size and quality of screen display, to avoid any bias caused by hardware.

7.1 User Study Design

We invited 20 participants to our user study. Out of 20, a total of 5 (25%) were female. The participants were drawn primarily from undergraduate and graduate students, with the age ranges between 20-35. The participants were sensitized about the amount of power saved and were informed that they do not have access to power.

Task-oriented study: We conducted a task-oriented study to assess whether FlexDisplay has any impact on users' efficiency or speed in completing their daily activities, including tasks such as shopping and entertainment. In this study, the participants were asked to complete eight predetermined tasks on a set of apps on both smartphones. The tasks are designed to simulate daily activities, such as shopping activity, e.g., placing designated items on the shopping cart on a shopping app, more details are summarized in Table 2. Each task should take around one minute to complete. To encourage active and patient participation, each participant received a $7 shopping credit upon completing the study.

Assumptions: We acknowledge that our user study makes a few assumptions about demography and user behavior. First, although our users are drawn primarily from students, we assume that they represent the behavior of all smartphone users. Second, the users are requested to follow their behavior of normal use, when they do not have access to a ready source of power. We assume that users, in being so instructed, follow their usual behavior. Finally, although we explicitly inform the users that their data is anonymized, we recognize the risk of users giving higher scores to appear agreeable. We mitigate this risk by explicitly highlighting the use of honest feedback, and further by asking the users for suggestions for further improvement.

To mitigate the influence of short-term memory and ensure a fair comparison, we designed a study specifically aimed at minimizing short-term memory impact. This is crucial as participants memorizing the initial batch of tasks could lead to consistently faster performance in repeated tasks. Specifically, we shuffled the 20 participants and divided them into two groups. The first group, with 10 randomly chosen participants, was assigned to perform all tasks on the first smartphone (with the original app versions). After a five-minute interval, they repeated the same tasks on the second smartphone, which featured the FlexDisplay-mode apps. In the second group, the sequence of smartphone usage was reversed. Participants initially performed all tasks using the second phone (FlexDisplay-mode apps) and subsequently attempted the tasks again on the first phone (original apps). This experimental design aimed to mitigate the influence of short-term memory, given that short-term memory is generally considered to last around 30 seconds [21]. Researchers recorded the time taken by each user to complete individual tasks using a stopwatch.

Questionnaire study: Participants were invited to share open-ended comments about the situations in which they would favor using the applications after completing the tasks above. Additionally, participants were also asked to played with all fifteen FlexDisplay-mode apps, and then rated their quality of experience with FlexDisplay on a scale of 1 to 5, with 5 representing the highest satisfaction level.

7.2 Results

7.2.1 Task-oriented Result. As shown in Figure 18, the results for completion times on those eight tasks indicate no significant time difference in task completion.

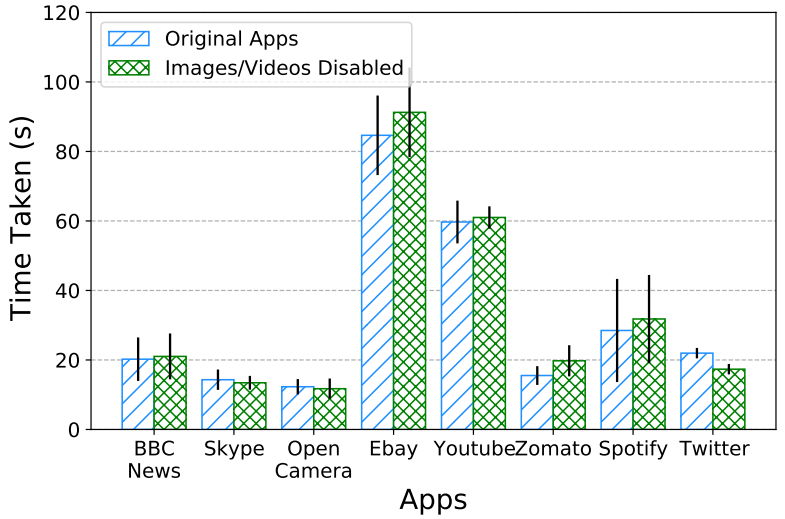


Fig. 18. Comparison of average duration taken to complete each particular task between original apps and FlexDisplay-mode apps, i.e., Images/Videos Disabled.

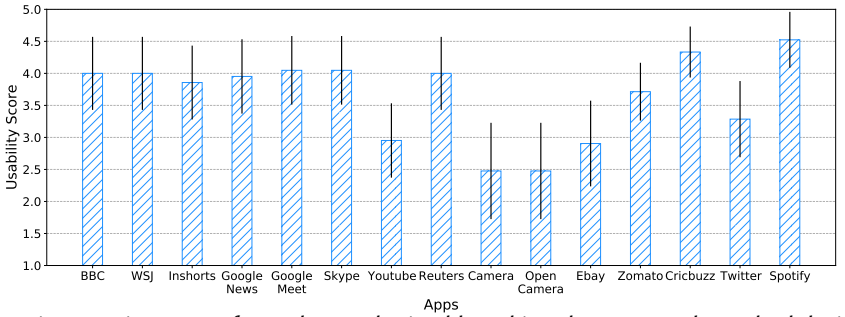


Fig. 19. Experience rating scores for each app obtained by taking the mean and standard deviation of 20 participants, the higher the better. (Y-axis starts from 1.)

The difference in the mean times taken for participants to perform the tasks per app was generally within a few seconds, with the largest difference being in Ebay (6.6 seconds) and the smallest difference being in open camera (0.56 seconds), with the average mean time being 2.116 seconds. This is only equal to 1.3% more than the time taken to perform on the original app, indicating that the differences are within the margin of error.

Conducting a paired t-test on each distribution, we found that only the increase in time for the Zomato app was statistically significant, while the others were statistically insignificant. From the result, it turns out that FlexDisplay does not impede users in performing these actions when images and videos are disabled.

7.2.2 Questionnaire Result. Figure 19 presents the rating scores from all participants when they used FlexDisplay-mode apps. The average quality of experience score across all participants and apps is *3.49*, suggesting a generally positive reception. Notably, participants expressed a favorable impact of FlexDisplay on BBC News and Spotify, with mean usability scores surpassing *4.0*. This preference was primarily attributed to participants deeming the disabled content as less crucial to their overall experience. Conversely, Open Camera received a slightly lower usability score,

just below *2.5*. For the remaining apps, the usability scores exceeded *3.0*, indicating that users are inclined to use FlexDisplay in various scenarios.

We also asked participants if they would prefer using FlexDisplay in any specific scenario or situation. Participants specified a wide range of scenarios, but a few that were common are:

- When the battery is low, and you need to perform a task.
- When you are traveling on a long journey and are anxious about your phone's battery life.
- When the app you are using does not need images or videos and can function properly with only text.

We further ask a few general questions to the participants. These include the questions:

(1) If participants liked this feature on any particular category of apps. Most participants specified the category "News" as the one where they strongly preferred this feature.
(2) If there are apps that are not part of this study where they would like the feature. Only 13 participants were able to come up with one or more apps, with the candidates named including taxi booking, messaging, ticket booking, and payment. Those apps are noted and we will consider scaling to those apps in our future work.
(3) Whether they liked the overall idea of using FlexDisplay on a scale of 1 to 5: The mean, median, and modes received were *3.68*, *3.9*, *4.0*, respectively, indicating that most participants liked the overall experience. However, a couple of participants (9.5%) also assigned a score of *2.0*, indicating that they are unlikely to utilize FlexDisplay. These participants on asked, preferred to utilize alternative techniques such as carrying power banks to avoid such power optimizations.

8 RELATED WORK

We classify prior works into two categories – works that optimize apps to reduce energy consumption and frameworks to optimize the displayed content.

Optimization of Apps: A number of techniques have been tried to reduce the energy consumption of individual apps. These include strategies such as offloading of compute-intensive strategies to other devices, as proposed by MAUI [11], CloneCloud [9] and Neurosurgeon [20]. A second technique is to identify common code hotspots and optimize them. For example, [24] and [31] specifically optimize HTTP requests and Java collections respectively, both of which are widely used in apps. Chimera [10] and EnergyPatch [4] take this strategy further by utilizing an automated way of identifying the energy hotspots and refactoring them in the wild. Our work builds upon these approaches, but specifically focuses on the energy consumption due to the computation related to rendering of content on display. GearDVFS [26] identifies the workload of the apps, and accordingly scales the frequency of the phone's processors. The work [8] jointly decides the frequencies of CPU and GPU to further conserve power. These techniques are orthogonal to our technique, and could potentially be built along with our changes.

Optimization of Display: As display consumed a significant portion of the total energy, a number of studies tried to optimize it. For example, [22] and [27] 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 [12]. For example, brightness dimming [18, 36, 40] is available on almost all modern smartphones to reduce energy consumption. ULPM [39] extends it by allowing users to interact without displaying any content on the smartphone display. Focus [37] reduces the brightness based on the importance of different content to users. FingerShadow [6] darkens pixels close to the user's fingers during their interaction as these pixels are not perceived by the users. ShutPix [41] develops a library that can reduce the density of pixels of some apps. The work [19] adapts the brightness to the ambient light present. FlexDisplay

utilizes similar strategies, but goes further than these studies by also disabling the rendering to save significantly more energy.

A few additional works also target the display content of specific apps. For example, Flash [5] identifies the important content when users scroll through web pages. Peo [34] optimizes the brightness level of smartphone display based on the video streamed to the user. LpGL [7] optimizes the brightness levels for virtual reality content. Chameleon [13] designs a color-adaptive web browser that changes the colors of the rendered web pages to conserve energy. RAVEN [17] regulates the frame rate of mobile games by looking at their perceptual similarity. These optimization techniques are orthogonal to that of FlexDisplay.

9 DISCUSSIONS

Through the extensive evaluation of FlexDisplay, we found it to be effective in saving the power of different categories of apps. Further, we discuss some of the major observations and limitations observed during our experiments.

Generalizability Across Applications: While we have evaluated FlexDisplay on a total of 8 app categories, it is possible to also utilize it on a few more different types of apps. This is because the technique used to identify the user interface elements is general in nature. Furthermore, it is relatively easy to add additional classes to the config file, which can be handled by FlexDisplay app. Beyond these two steps, there is no app-specific strategy used in FlexDisplay. We have not evaluated FlexDisplay on any game app, any app that provides delivery or location-based services and productivity apps. We expect FlexDisplay to generalize to most of these apps, with one exception. Currently, FlexDisplay does not work on location-based apps where *maps* is the most important user interface component. This is because such apps bypass the *View* framework layer for rendering maps. It directly makes a call to the Android runtime API for location. Hence, FlexDisplay cannot be used as currently on such apps. In future, we plan to extend the support to these types of apps as well.

Additional Savings via Disabling Fetching of Content: A current limitation of FlexDisplay is that it only disables rendering of content, while still fetching them over the network. Fetching such content that are not utilized can also lead to wastage of both power and bandwidth. We plan to disable such fetching of content as part of our future work.

Requires Rooting of Smartphones: The current iteration of FlexDisplay necessitates rooting of the smartphone, as this allows for the installation of the kernel and the modified View system. However, our long-term vision entails a system akin to FlexDisplay being incorporated as an additional feature by device manufacturers. Additionally, for this study, we exclusively focused on Android smartphones, as they are open-sourced and straightforward to upgrade. We anticipate that integrating with iOS smartphones would involve a similar process, given their utilization of a similar user interface system.

Disabling of Banner Advertisements: FlexDisplay intentionally refrains from disabling in-app advertisements, even though the rendering of banner ads consumes a significant amount of power in some cases. This decision is made considering that many app developers rely on these advertisements to sustain their income sources for app development. Therefore, we have chosen not to disable advertisements, although it is technically feasible to do so using FlexDisplay.

Possibility of Disabling Required Prompts: While FlexDisplay has been designed with the objective of disabling user interface elements that are unnecessary, we acknowledge that occasionally it might disable needed interaction. For example, disabling images in a news app might have the unintended consequence of also disabling the interactive response system occasionally present in news articles. In case of camera, the annotations that aid the users in picking the right angle also get disabled. To account for such factors, we explicitly included these cases in the user study,

and informed users among them. The vast majority of users (as shown in user study) agreed that the benefits of power saving outweigh the disadvantages of occasionally losing out some prompts. In most cases, users recognized when prompts were missed, and they used FlexDisplay's app to render the user interface specifically for that app.

Requirement of Manual Effort: One challenge of using FlexDisplay is that it requires some amount of manual effort to identify the user interface elements, though this effort is mitigated by giving hints using our heuristic. While we acknowledge that FlexDisplay requires such identification for each app, we also note that the user interface names do not change across versions of the apps. Thus, this is a one-time effort for each app. In our experience, this effort takes less than 10 minutes per app, making the amount of manual effort minimal. Furthermore, we also recognize that additional automation using machine learning techniques are possible, which we leave for future work. It is also feasible to have a swiping gesture enabled to enable or disable an individual app for easier use.

Failing to Identify Correct User Interface Element: Since our technique of identifying user interface elements is a heuristic, it is possible that it fails to identify the right user interface element. Note that because this step of identifying the user element has a human-in-the-loop, we assume that such developers would identify the user interface elements correctly. Although we have not encountered any app among the ones we have tried FlexDisplay where our heuristic failed, we recognize that in case it occurs, FlexDisplay will not disable any user interface elements, and will, therefore, not save any power.

10 CONCLUSION

Low battery anxiety is a significant concern among smartphone users, one reason is due to the inefficiency of current mobile systems and applications in displaying the UI. These systems often lack flexible options for users to stop displaying unnecessary or unwanted UI elements. In this paper, we propose FlexDisplay, which provides users with easy customization options to selectively disable UI elements with minimal disruption to the user experience. FlexDisplay also prevents users' mis-taps on disabled UI regions to avoid confusion and negative user experience. Achieving these goals involves upgrading the Android software stack, including the View system at the framework layer and the Android kernel layer. Designed to support numerous apps across various smartphones, FlexDisplay demonstrates significant power savings and usability improvements across 15 apps spanning 8 different genres for two different smartphone models. Our results consistently show over 10% power savings (up to 47.47%). User studies indicate that FlexDisplay does not hinder users' ability to perform tasks within apps, with a median user rating of 3.8/5, reflecting its favorable usability score.

REFERENCES

[1] [n. d.]. Monsoon Power Monitor. https://www.msoon.com/high-voltage-power-monitor. Accessed on May 24, 2023.

[2] [n. d.]. Synaptics DSX touchscreen driver. https://github.com/synaptics-touch/synaptics-dsx-i2c. Accessed on May 24, 2023.

[3] Brooke Auxier. July 28, 2022. Shuffle, subscribe, stream: Consumer audio market is expected to amass listeners in 2024, but revenues could remain modest. https://www2.deloitte.com/xe/en/insights/industry/technology/technology-media-and-telecom-predictions/2024/consumer-audio-market-trends-predict-more-global-consumers-in-2024.html Accessed on Mar 8, 2023.

[4] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Transactions on Software Engineering* 44, 5 (2018), 470–490. doi:10.1109/TSE.2017.2689012

[5] Hao-Chun Chang, Yu-Chieh Yang, Liang-Yan Yu, and Chun-Han Lin. 2019. FLASH: Content-based Power-saving Design for Scrolling Operations in Browser Applications on Mobile OLED Devices. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. doi:10.1109/ISLPED.2019.8824875

[6] Xiang Chen, Kent W. Nixon, Hucheng Zhou, Yunxin Liu, and Yiran Chen. 2014. FingerShadow: An OLED Power Optimization Based on Smartphone Touch Interactions. In *Proceedings of the 6th USENIX Conference on Power-Aware Computing and Systems* (Broomfield, CO) *(HotPower'14)*. USENIX Association, USA, 6.

[7] Jaewon Choi, HyeonJung Park, Jeongyeup Paek, Rajesh Krishna Balan, and JeongGil Ko. 2019. LpGL: Low-Power Graphics Library for Mobile AR Headsets. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. 155–167. doi:10.1145/3307334.3326097

[8] Yonghun Choi, Seonghoon Park, and Hojung Cha. 2019. Graphics-aware Power Governing for Mobile Devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. 469–481. doi:10.1145/3307334.3326075

[9] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. 301–314. doi:10.1145/1966445.1966473

[10] Marco Couto, João Saraiva, and João Paulo Fernandes. 2020. Energy Refactorings for Android in the Large and in the Wild. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 217–228. doi:10.1109/SANER48275.2020.9054858

[11] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (San Francisco, California, USA) *(MobiSys '10)*. 49–62. doi:10.1145/1814433.1814441

[12] Pranab Dash and Y. Charlie Hu. 2021. How Much Battery Does Dark Mode Save? An Accurate OLED Display Power Profiler for Modern Smartphones. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. 323–335. doi:10.1145/3458864.3467682

[13] Mian Dong and Lin Zhong. 2012. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *IEEE Transactions on Mobile Computing* 11, 5 (2012), 724–738. doi:10.1109/TMC.2012.40

[14] Anshak Goel, Deeptorshi Mondal, Manavjeet Singh, Sahil Goyal, Navneet Agarwal, Jian Xu, Mukulika Maity, and Arani Bhattacharya. 2024. FlexDisplay: A Flexible Display Framework To Conserve Smartphone Battery Power. In *2024 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 401–404. doi:10.1109/PerComWorkshops59983.2024.10502958

[15] Lui Gough. [n. d.]. Experiment: microSD Card Power Consumption & SPI Performance. https://goughlui.com/2021/02/27/experiment-microsd-card-power-consumption-spi-performance/ Published on Feb 27, 2021; Accessed on Jan 10, 2024.

[16] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. 2022. A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2879–2904. doi:10.1109/TSE.2021.3071193

[17] Chanyou Hwang, Saumay Pushp, Changyoung Koh, Jungpil Yoon, Yunxin Liu, Seungpyo Choi, and Junehwa Song. 2017. RAVEN: Perception-aware Optimization of Power Consumption for Mobile Games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. 422–434. doi:10.1145/3117811.3117841

[18] Samuel Isuwa, David Amos, Amit Kumar Singh, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2023. Content- and Lighting-Aware Adaptive Brightness Scaling for Improved Mobile User Experience. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–2. doi:10.23919/DATE56975.2023.10136915

[19] Samuel Isuwa, David Amos, Amit Kumar Singh, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2023. Maximising mobile user experience through self-adaptive content- and ambient-aware display brightness scaling. *Journal of Systems Architecture* 145 (2023), 103023. doi:10.1016/j.sysarc.2023.103023

[20] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. 615–629. doi:10.1145/3037697.3037698

[21] Marco Cascella; Yasir Al Khalili. [n. d.]. Short-Term Memory Impairment. https://www.ncbi.nlm.nih.gov/books/NBK545136/ Accessed on July 17, 2023.

[22] Dongwon Kim, Nohyun Jung, and Hojung Cha. 2014. Content-Centric Display Energy Management for Mobile Devices. In *Proceedings of the 51st Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '14)*. 1–6. doi:10.1145/2593069.2593113

[23] Jaeheon Kwak, Sunjae Lee, Dae R. Jeong, Arjun Kumar, Dongjae Shin, Ilju Kim, Donghwa Shin, Kilho Lee, Jinkyu Lee, and Insik Shin. 2023. MixMax: Leveraging Heterogeneous Batteries to Alleviate Low Battery Experience for Mobile Users. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services* (Helsinki, Finland) *(MobiSys '23)*. 247–260. doi:10.1145/3581791.3596843

[24] Ding Li, Yingjun Lyu, Jiaping Gui, and William G. J. Halfond. 2016. Automated Energy Optimization of HTTP Requests for Mobile Applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 249–260. doi:10.1145/2884781.2884867

[25] Yang Li, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Feng Qian, Jing Yang, Hao Lin, Yunhao Liu, Bo Xiao, Xiaokang Qin, and Tianyin Xu. 2025. Dissecting and Streamlining the Interactive Loop of Mobile Cloud Gaming. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 595–611. https://www.usenix.org/conference/nsdi25/presentation/li-yang

[26] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. 2023. *A Workload-Aware DVFS Robust to Concurrent Tasks for Mobile Devices*. doi:10.1145/3570361.3592524

[27] Mihai Matei. July 28, 2022. Samsung patents new energy-efficient display refresh rate technology. https://www.sammobile.com/news/samsung-patents-new-energy-efficient-display-refresh-rate-technology/ Accessed on Mar 8, 2023.

[28] John Moreno. May . YouTube Is The Top Preferred Platform For Podcasts. https://www.forbes.com/sites/johanmoreno/2022/05/27/youtube-is-the-top-preferred-platform-for-podcasts/ Accessed on May 20, 2023.

[29] Wojciech Mrozik, Mohammad Ali Rajaeifar, Oliver Heidrich, and Paul Christensen. 2021. Environmental impacts, pollution sources and pathways of spent lithium-ion batteries. *Energy Environ. Sci.* 14 (2021), 6099–6121. Issue 12. doi:10.1039/D1EE00691F

[30] Sagar Naresh. 23 April 2023. Google Meet will let you now turn off other participant's video feed – SamMobile. https://www.sammobile.com/news/google-meet-turn-off-video-feed-participants/.

[31] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. 2019. Recommending Energy-Efficient Java Collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 160–170. doi:10.1109/MSR.2019.00033

[32] Daniele Jahier Pagliari, Santa Di Cataldo, Edoardo Patti, Alberto Macii, Enrico Macii, and Massimo Poncino. 2021. Low-Overhead Adaptive Brightness Scaling for Energy Reduction in OLED Displays. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2021), 1625–1636. doi:10.1109/TETC.2019.2908257

[33] Pijush Kanti Dutta Pramanik, Nilanjan Sinhababu, Bulbul Mukherjee, Sanjeevikumar Padmanaban, Aranyak Maity, Bijoy Kumar Upadhyaya, Jens Bo Holm-Nielsen, and Prasenjit Choudhury. 2019. Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-Art Review of Smartphone Battery and Energy Usage. *IEEE Access* 7 (2019), 182113–182172. doi:10.1109/ACCESS.2019.2958684

[34] Chao Qian, Daibo Liu, and Hongbo Jiang. 2022. Harmonizing Energy Efficiency and QoE for Brightness Scaling-based Mobile Video Streaming. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*. 1–10. doi:10.1109/IWQoS54832.2022.9812899

[35] Chatura Samarakoon, Gehan Amaratunga, and Phillip Stanley-Marbell. 2021. Content-Aware Automated Parameter Tuning for Approximate Color Transforms. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services* (Oldenburg, Germany) *(MobileHCI '20)*. Article 14, 6 pages. doi:10.1145/3406324.3410713

[36] Donghwa Shin, Younghyun Kim, Naehyuck Chang, and Massoud Pedram. 2013. Dynamic Driver Supply Voltage Scaling for Organic Light Emitting Diode Displays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 7 (2013), 1017–1030. doi:10.1109/TCAD.2013.2248193

[37] Kiat Wee Tan, Tadashi Okoshi, Archan Misra, and Rajesh Krishna Balan. 2013. FOCUS: A Usable and Effective Approach to OLED Display Power Management. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Zurich, Switzerland) *(UbiComp '13)*. 573–582. doi:10.1145/2493432.2493445

[38] Dimira Teneva. [n. d.]. Repeat purchase rate for ecommerce brands. https://www.metrilo.com/blog/repeat-purchase-rate Accessed on May 20, 2023.

[39] Jian Xu, Suwen Zhu, Aruna Balasubramanian, Xiaojun Bi, and Roy Shilkrot. 2018. Ultra-Low-Power Mode for Screenless Mobile Interaction. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. 557–568. doi:10.1145/3242587.3242614

[40] Zhisheng Yan and Chang Wen Chen. 2016. RnB: Rate and Brightness Adaptation for Rate-Distortion-Energy Tradeoff in HTTP Adaptive Streaming over Mobile Devices. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking* (New York City, New York) *(MobiCom '16)*. 308–319. doi:10.1145/2973750.2973780

[41] Zhisheng Yan and Chang Wen Chen. 2017. Too Many Pixels to Perceive: Subpixel Shutoff for Display Energy Reduction on OLED Smartphones. In *Proceedings of the 25th ACM International Conference on Multimedia* (Mountain View, California, USA) *(MM '17)*. 717–725. doi:10.1145/3123266.3123344

[42] Yu Zhang, Guoming Tang, Qianyi Huang, Kui Wu, Yangjing Wu, and Yi Wang. 2022. Investigating Low-Battery Anxiety of Mobile Users. In *2022 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. 272–279. doi:10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics55523.2022.00022