D-VSync: Decoupled Rendering and Displaying for Smartphone Graphics

Yuanpei Wu[1,2], Dong Du[1,2], Chao Xu[3], Yubin Xia[1,2], Ming Fu[3], Binyu Zang[1,2], Haibo Chen[1,2,4]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

[2]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education

[3]Fields Lab, Huawei Central Software Institute

[4]Key Laboratory of System Software (Chinese Academy of Science)

Abstract

Rendering service, which typically orchestrates screen display and UI through Vertical Synchronization (VSync), is an indispensable system service for user experiences of smartphone OSes (e.g., Android, OpenHarmony, and iOS). The recent trend of large high-frame-rate screens, stunning visual effects, and physics-based animations has placed unprecedented pressure on the VSync-based rendering architecture, leading to higher frame drops and longer rendering latency.

This paper proposes Decoupled Vertical Synchronization (*D-VSync*), which decouples execution and displaying in the rendering service. *D-VSync* allows frames to be rendered a number of VSync periods before being physically displayed on the screen. The key insight behind *D-VSync* to resolve the limitation of VSync is that, *the decoupling enables sporadic long frames to utilize the computational power saved by common short frames, therefore providing a larger time window to tolerate workload fluctuations.* Evaluation results of 75 common OS use cases and apps on OpenHarmony (Mate 40 Pro, Mate 60 Pro), 25 popular apps on Android (Google Pixel 5), and simulations of 15 mobile games show that compared to VSync, *D-VSync* on average reduces frame drops by 72.7%, user-perceptible stutters by 72.3%, and rendering latency by 31.1%, with only 0.13%–0.37% more power consumption. *D-VSync* has been integrated into HarmonyOS NEXT.

CCS Concepts: **• Software and its engineering → Operating systems**; **• Computing methodologies → Graphics systems and interfaces**.

Keywords: Operating system, graphics system, rendering service, vertical synchronization

ACM Reference Format:
Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Ming Fu, Binyu Zang, Haibo Chen. 2025. D-VSync: Decoupled Rendering and Displaying for Smartphone Graphics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3669940.3707235

1 Introduction

Smartphones have increasingly penetrated into people's everyday life, resulting in an average usage of over 4.6 hours per day for non-voice activities [43]. A majority of services offered by smartphones heavily involve graphics UI (GUI), where a performant rendering service is indispensable for satisfiable user experiences [37, 62, 65].

Modern smartphone rendering architectures mainly stem from the Vertical Synchronization (VSync) with triple buffering from Google's Project Butter [39] dated back to 2012. The goal of VSync is to synchronize the app rendering speed with the screen refresh rate, through a pipeline design where different steps like app logic, rendering, surface compositing, and frame display all happen synchronously when a screen VSync signal is triggered (§2). Android [13], iOS [20], OpenHarmony [23], and browser engines [3] all adopt the VSync rendering architecture [4, 10, 22, 24] for mobile devices.

However, there is a challenge for such an architecture with the advent of high-resolution, high-refresh-rate, and foldable screens. Specifically, the number of pixels that the rendering service needs to render per second has increased about 25 times compared to the original iPhone 4 and Galaxy S. Even worse, advanced visual effects and physics-based animations, such as Gaussian blur, dynamic shadows, and particle effects, create substantial loads at specific key frames (§3.1). Such a substantial increase of load, which far exceeds the increasing pace of silicon advances, may cause harmed user experiences such as user-perceptible stutters.

To illustrate the challenge, the paper presents a deep analysis on contemporary rendering services from the perspective of leading smartphone vendors. First, *smartphone OSes still face severe frame drops.* Specifically, a frame drop occurs when the rendering fails to keep pace with the screen refreshing, causing the previous content to be displayed again for another frame. Our quantitative evaluation (with industrial criteria) for 75 common use cases show that 26.6% and 38.6% cases (20 and 29) exhibited frame drops with an average frame drops per second (FDPS) of 7.51 and 8.42 with GLES [11] and Vulkan [25] backends, on state-of-the-art commercial smartphone (Mate60 Pro, 120Hz screen, OpenHarmony [23]). Many important cases like closing the notification center or clearing all notifications can only reach 95–105 FPS on the 120 Hz screen, causing noticeable stutters to users. More experiments on different devices confirmed the prevalence of frame drops in existing smartphone usages (§3.2). The main reason is that frame drops are hard to eradicate under VSync, where heavy long frames cannot always complete before the fixed VSync deadline.

One intuitive approach is to leverage triple buffering in VSync. This, however, *yields high rendering latency*: the time difference between the timestamp represented by the frame content and the time when this frame is displayed on the screen. When a frame drop happens, subsequent rendered buffers will be "stuffed" into the third slot of the buffer queue, waiting for an extra VSync period. We measured an average of 45.8 ms end-to-end rendering latency on Pixel 5, and 32.2 ms and 24.2 ms on Mate 40 Pro and Mate 60 Pro, respectively (§3.3). Yet, prior studies suggested that lags of responsiveness impact more quality of experience (QoE) than frame drops [61], and the Just Noticeable Difference (JND) for human-eye latency discrimination is ≤15 ms [53, 54].

Based on our long-term experience in rendering services and the analysis of real-world traces, we discovered **the power law distribution of frame rendering time**: the majority (≥95%) of frames are short and quick while a small portion (≤5%) of key frames being heavily-loaded that cause frame drops (Figure 1). The root cause of frame drops and long rendering latency lies in *the presence of bursty heavily-loaded long frames.* Such key frames need to process complex application or rendering logic and fail to complete within its designated VSync period, leading to stutters. The VSync rendering architecture forcibly synchronizes the timing of rendering with the fixed timing of screen refreshing, and thus the core conflict is the *always-fixed display refresh period* versus the *fluctuation of rendering workloads* with long and short frames.

Based on this observation, this paper presents Decoupled Vertical Synchronization (*D-VSync*), a novel rendering architecture in smartphone OSes addressing the issues of frame drops and rendering latency. The key insight of *D-VSync* is to break *the close coupling of the rendering execution and the periodic VSync events. D-VSync* addresses the limitations of VSync by performing rendering executions in advance under possible and deterministic scenarios through a *decoupled rendering and displaying* design. *D-VSync* allows sporadic long frames to utilize the time saved by common short frames, thus providing a larger time window to withstand stutters caused by workload fluctuations.

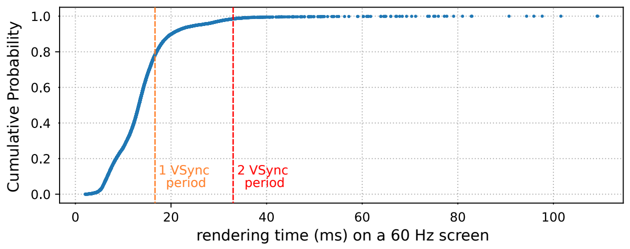


Figure 1. Cumulative Distribution Function (CDF) of the frame rendering time (following the power law distribution). Most frames finish in one VSync period (78.3%). However, despite the support of triple buffering, approximately 5% of frames fail to finish on time, causing stutters.

The design of *D-VSync* faces three challenges. First, *D-VSync* can no longer use the VSync signal of the screen to trigger each frame's rendering logic at a specific frequency. Instead, *D-VSync* needs to explicitly control the timing, pacing, pre-rendering limit, and frame buffer usage of each frame, based on different scenarios and requirements. To address this challenge, *D-VSync* proposes *Frame Pre-Executor (FPE)*. When the *FPE* module determines that pre-rendering is feasible, during deterministic animations and simple interactions, it sends the Decoupled-VSync signal to trigger frame execution of the related rendering processes (the app process and the rendering service), ahead of the screen's VSync signal to fully utilize the idle CPU time saved by short frames.

Second, *D-VSync* needs to ensure that all the content rendered are totally correct, even though some dependencies of a frame may not be ready during decoupled pre-rendering. For animations, every frame needs correct timing to sample motion curves for proceeding the dynamic effects. As *D-VSync* enables animations to run forward from the present, simply using the current timestamp is not sufficient. Therefore, *D-VSync* proposes *Display Time Virtualizer (DTV)* to tackle this challenge. *DTV* virtualizes the future display time of the rendering from the current execution time of the code, enabling apps and the rendering framework to foresee when the frame will be displayed. Consequently, frames can use the virtualized display time to render its content.

Finally, *D-VSync* should be extensible to various user interaction scenarios and custom-rendering apps that bypass the OS rendering framework (e.g., games or browsers). The new abstractions and the high-performance mode make compatibility a challenge. *D-VSync* thereby proposes *dual-channel decoupling APIs*: regular native apps get default performance and frame rate benefits from *D-VSync* in the decoupling-oblivious channel without any source code modifications needed. Furthermore, *D-VSync* offers custom interfaces for decoupling-aware apps, which achieve a balance between exceptional performance and compatibility. We further carefully design *D-VSync* to be compatible with the latest technologies such as LTPO dynamic frame rates [31].

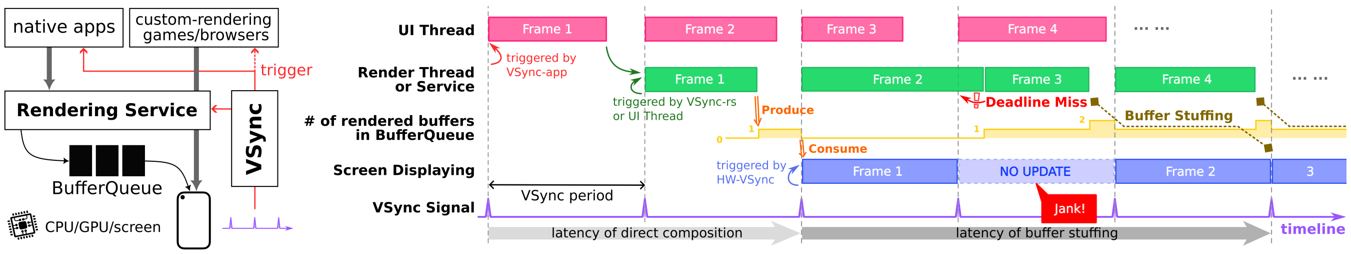


Figure 2. VSync rendering architecture and its rendering pipeline. In the runtime trace, Frame-2 is heavily-loaded and misses the VSync deadline, causing a frame drop. Subsequent frames experience a higher latency due to buffer stuffing.

We have implemented *D-VSync* on OpenHarmony 4.0 and Android 13. Comprehensive evaluations are conducted on both the baseline systems and the *D-VSync*-based systems on Google Pixel 5, Mate 40 Pro, and Mate 60 Pro. Our evaluations encompass microbenchmarks, 75 common OS use cases, 25 popular real-world applications, 15 mobile game simulations, and 2 case studies focusing on decoupling-aware applications. The experiments are conducted both automatically under production testing frameworks and manually by professional user experience (UX) evaluators for objective data and subjective data, respectively. The results demonstrate significant improvements achieved by *D-VSync*. Specifically, on average, *D-VSync* reduces the number of frame drops by 72.7%, user-perceptible stutters by 72.3%, and rendering latency by 31.1%, with 0.13%–0.37% increase in power consumption. *D-VSync* has been integrated into HarmonyOS NEXT as a key feature for smoothness.

2 VSync Rendering Architecture

This section introduces the basic concepts of VSync architecture and the specific designs on smartphone OSes.

Buffer queue. A frame buffer stores all the pixel data and metadata of a frame, and a buffer queue is a First-In First-Out (FIFO) queue of frame buffers, maintaining a producer-consumer model of frames. Broadly speaking, the producer of the model is the rendering service that renders frames, and the consumer is the screen that displays frames, as shown by the architecture diagram in Figure 2. A buffer queue contains one *front buffer* for the screen panel to refresh its pixels, and one or more *back buffers* for the software rendering architecture to render into. Android and iOS configure two back buffers (i.e., triple buffering), while OpenHarmony uses three so that consecutive frames can render in parallel.

VSync signals. The swap between front-buffer and back-buffer happens during the VSync signals (the purple trace in Figure 2). Smartphone screens update frames at a configured refresh rate, e.g., 60 Hz (or 120 Hz). Before every physical panel refresh, the screen generates a hardware VSync signal, marked HW-VSync, sending it to the rendering architecture via a hardware abstraction layer (HAL) every fixed 16.7 ms (or 8.3 ms). If the swap-in is not aligned with HW-VSync signals, the display may show content from two frames, resulting in screen tearing [59]. This also means that the rendering of a frame must finish before the VSync deadline, otherwise the screen panel will have to keep the last frame for another VSync period with nothing to update, known as a jank (i.e., frame drop).

VSync pipeline. The end-to-end rendering procedure usually spans at least two VSync periods, following a pipeline structure illustrated in Figure 2. Different pipeline stages such as the app UI logic, rendering, buffer swap-in, are triggered by different software VSync signals (usually with fixed offsets from HW-VSync). Specifically, in the first step, the VSync-app signal will trigger the App UI thread to handle inputs and deal with the UI logic. Upon completion, the UI thread will invoke the render thread in Android, or the render service in OpenHarmony/iOS triggered by the VSync-rs signal, to deal with animations and GPU rendering. Finally, the buffer swap-in is handled by the SurfaceFlinger [9] in Android triggered by the VSync-sf signal, or a designated hardware thread in OpenHarmony/iOS aligning with HW-VSync (omitted in Figure 2 for simplicity). To summarize, the VSync rendering architectures use the consumer VSync signals to drive each stage of the frame production in the pipeline, to ensure every frame starts timely in real time.

Challenges of nowadays VSync architectures. However, the VSync rendering architecture cannot guarantee that each stage of the pipeline can finish within its designated VSync period. The computational power for mobile devices does **not have much redundancy to tolerate workload fluctuations**, which are increasingly common nowadays as the demands for visual quality rise (§3.1). The trace in Figure 2 illustrates a concrete example of the frame drop and rendering latency. If a workload-heavy frame unfortunately misses the VSync deadline (Frame-2), then the screen will have nothing to update, resulting in a jank. Users may experience a stutter if it is a key frame in a series of screen updates. Unfortunately, such frame drops are not uncommon. Even worse, a frame drop lengthens the latency of subsequent frames. For Frame-1, the rendering latency is the time difference from when the UI thread starts (triggering of VSync-app) to when the frame gets displayed, marked in the light gray arrow, about 2 VSync periods. Because the display of Frame-2 defers, the rendered buffer from Frame-3 will be stuffed in the buffer queue and wait for another unnecessary VSync

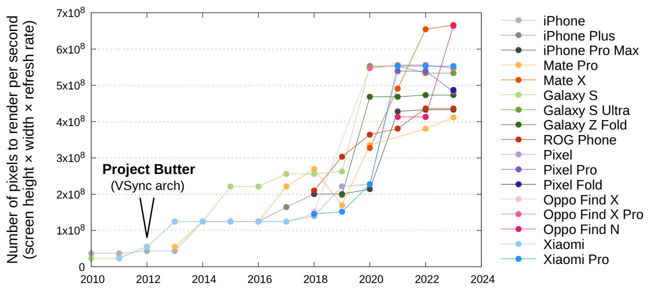


Figure 3. Trend in the number of pixels rendered per second by the rendering architecture. The number has increased about 25 times over the past one and a half decades.

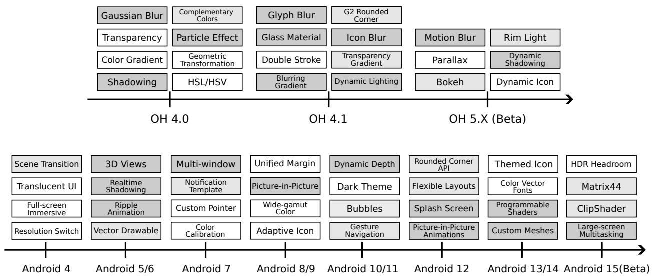


Figure 4. Trend in the increasing number of graphics features supported. The darker visual effects represent heavier rendering workloads in the key frames.

period. All subsequent frames experience a delay of 3 VSync periods, marked in the dark gray arrow, until another long frame emerges. Longer rendering latency results in more lag in the responsiveness of the system (§3.3).

3 Characterizing OS Rendering Service

Despite the critical importance of rendering services in modern smartphone OSes, the topic is often underestimated and (even) considered a resolved issue. This section presents an in-depth analysis (from the perspective of leading smartphone vendors) to show the current trends in rendering services, the inherent limitations of the VSync, and our insights. We believe the analysis will not only motivate this work, but also encourage a wide range of future research.

3.1 Trends in Rendering

Through summarizing the display hardware and OS rendering capabilities of top smartphone brands over the past decade, we observed clear trends in rendering: *smartphone vendors tend to deliver more immersive and richer visual experiences to meet the increasing demand for exquisite and attractive content from end users.*

First, the baseline workloads of rendering nowadays has increased about 25 times. Figure 3 illustrates the number of pixels that the smartphone OS has to render per second across flagship smartphones since the advent of the iPhone 4 and the first Galaxy S in 2010. Smartphone vendors are

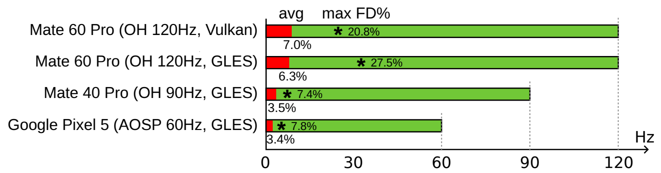


Figure 5. Summary of the average and maximum percentage of frame drops (FD) over the total display time.

integrating displays that are clearer (higher PPI, Pixels Per Inch), smoother (higher refresh rates), and more immersive (larger screen sizes) into their products. This trend continues to grow, as evidenced by the successful sales of foldable smartphones in 2023 and 2024 (Mate X, Galaxy Z, etc.) and the gradual production of 144 Hz and 165 Hz phone screens.

Second, real-world scenes involve different combinations of complex visual effects, making the rendering workloads fluctuating and unpredictable. Figure 4 makes a growing list of the supported graphics features in the rendering services since Android 4 and OpenHarmony 4.0. The darker effects represent heavier tasks where key frames require a substantial amount of work (usually over 1 ms), while subsequent frames may or may not be able to reuse the rendered cache.

The gap. Consequently, traditional VSync rendering architectures from over a decade ago are struggling to handle the increasing rendering workloads: the volatile and unstable frame execution time, along with the complicated logic required to process visual effects and animations, often fails to finish before the constant VSync deadline. Next, we present the statistics and analysis regarding the issues.

3.2 A Quantitative Analysis on Frame Drops

Frame drop refers to the case where rendering misses the VSync deadline and the screen has no buffer to update. It is one of the most important metrics to show the performance of rendering service and the user experience in industry.

Methodology and results. We first inspected 75 common OS use cases by a industrial testing framework simulating necessary user operations. A full frame rate is required for these cases in industrial criteria. However, on Mate 40 Pro (90 Hz screen) and Mate 60 Pro (120 Hz) with OpenHarmony and GLES backend [11], 9 and 20 out of 75 have frame drops, with an average frame drops per second (FDPS) of 3.17 and 7.51. In the current implementation of Vulkan backend [25], 29 cases have frame drops with 8.42 FDPS. Many important cases such as closing the notification center, clearing all notifications, etc., can only reach 95–105 FPS, which are noticeable to users and greatly affect the QoE of products. We also inspected altogether 23,000 frames of 25 different Android apps on Google Pixel 5 (60 Hz), by swiping the screen twice a second to let the app keep rendering new content. The average FDPS for those apps is 2.04, occupying 3.4% of the total display time. Figure 5 illustrates a summary, and the blue bars in Figure 11, 12, 13, and 14 present the details.

Based on our long-term experience with rendering and analysis of real-world traces [8], we conclude that the deep reason behind frame drops is the *workload fluctuation*. We discover *the power law distribution of frame rendering time*:

Workload fluctuation is an inherent characteristic of rendering tasks, with the majority (≥95%) of frames being short and quick, and a small portion (≤5%) of key frames being heavily loaded that ultimately determines the user experience.

Overall, current VSync architectures lack the capability of handling workload bursty and abdicate these responsibilities to graphics programmers, which we consider improper.

3.3 A Quantitative Analysis on Rendering Latency

Long rendering latency is another significant issue, further exacerbated by the buffer stuffing in VSync triple-buffering architectures. As discussed in §2, a frame drop causes all the subsequent frames to wait inside the third slot of the buffer queue for another unnecessary VSync period.

Methodology and results. Figure 6 demonstrates the distribution of frames for representative apps. Most frames get stuffed into the buffer queue rather than being directly composited because of the frequent frame drops. Besides, we measured an average rendering latency of 45.8 ms over all the rendering workloads we recorded on Google Pixel 5. On Mate 40 Pro and Mate 60 Pro, the latency averages at 32.2 ms and 24.2 ms, respectively. To directly visualize its consequence and impact on the responsiveness of the system, we write a simple app illustrated in Figure 7. It draws a red ball every frame at the position of the touch event. In the ideal situation without any latency, the ball should strictly follow the touch and be covered by the fingertip. However, as we swipe the finger upwards, we can clearly see the ball that falls behind. When the latency is 45 ms, the maximum position difference from the fingertip to the ball reaches around 400 pixels (2.4 cm) as we swipe fast.

The long rendering latency is still an open challenge for smartphones. Efforts on desktop AAA games like G-Sync [5], Reflex [6], and FreeSync [2] cannot solve the issue due to high power consumption and special hardware required [56].

3.4 Insights

The root cause of the issues is that: current rendering services assume that the content to be rendered corresponds to the present moment in time. Thus, the rendering execution must be placed just before the frame displays to avoid any lag, leaving it a very short time window to finish. Based on the observation of the power law distribution of rendering, our idea to tackle the issues is to **utilize the computational power saved by common short frames to provide a larger time window to tolerate workload fluctuations**.

To this end, our key insight is to break the assumption and *decouple the time that the rendering workload executes and the time that the rendering frame represents*. Under this premise, a frame can be rendered much earlier under possible and deterministic conditions such as animations and simple interactions, only with the knowledge about when the content will be displayed. The decoupled pre-rendering provides sporadic long frames a larger time window to withstand stutters, utilizing the idle time saved by common short frames. *D-VSync* (our system) enlarges the maximum number of buffers that the buffer queue can hold, so that if a long frame runs longer than expected, the screen is still able to consume the pre-computed short frames in the buffer queue.

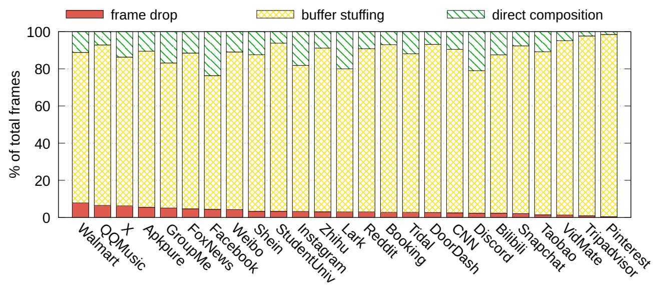


Figure 6. Distribution of frames. Most frames wait inside the buffer queue for another period due to buffer stuffing after frame drops, creating unnecessary latency.

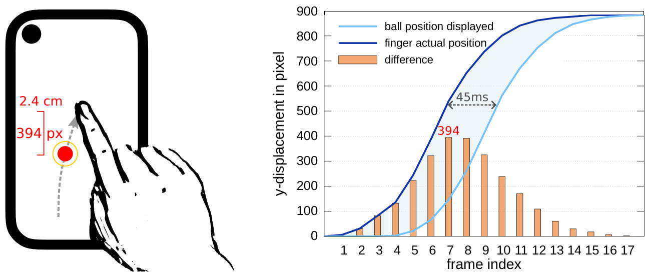


Figure 7. Visualization of the rendering latency. The ball falls behind the fingertip for at most 400 pixels.

To implement a decoupled rendering architecture, we face three challenges (detailed in §1): First, *D-VSync* must *explicitly manage the execution of individual frames* when pre-rendering is feasible under deterministic scenarios. Second, when rendering ahead, *D-VSync* needs to guarantee that *the frame content remains correct*, as it would under VSync architectures. Last, *D-VSync* should be *extensible and compatible with diverse interactive scenarios and custom-rendering applications* (e.g., games or browsers).

4 Decoupled Rendering and Displaying (*D-VSync*)

To tackle these challenges, we propose Decoupled Vertical Synchronization (*D-VSync*), a novel rendering architecture enhancing the original VSync for workload instability.

4.1 The Overall Design of *D-VSync*

Figure 8 illustrates the overall architecture of *D-VSync* in the OS rendering service. It offers another path for the frame

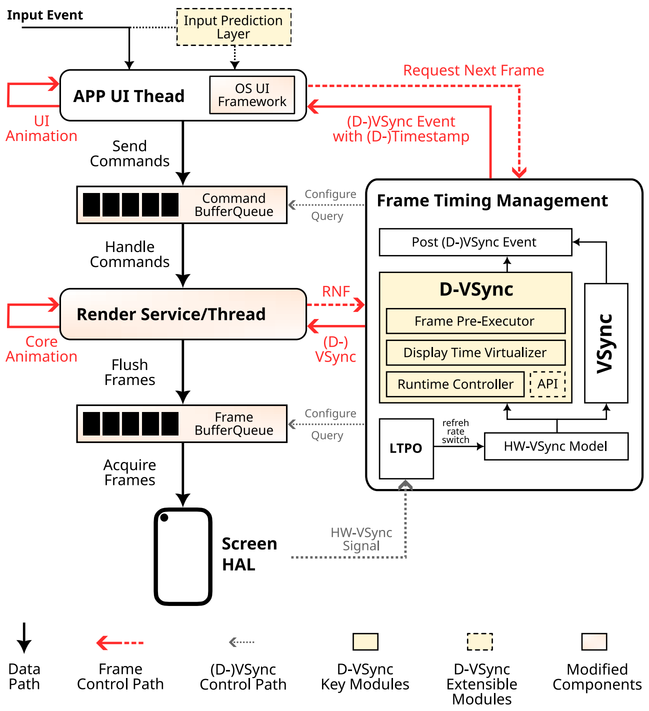


Figure 8. The *D-VSync* rendering architecture, enhancing the conventional VSync for workload fluctuations.

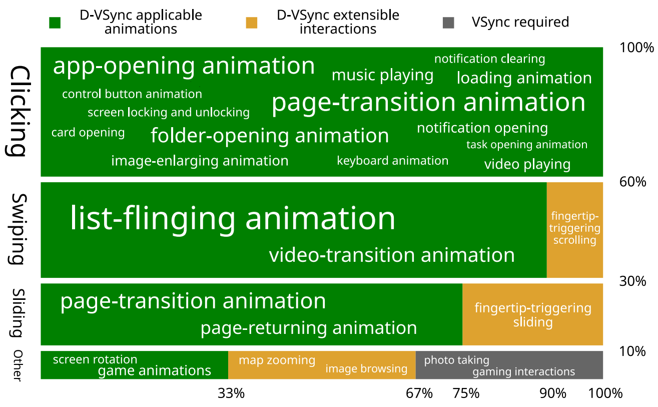


Figure 9. The scope of *D-VSync* approach. *D-VSync* is applicable to deterministic frames in animations (85%), and extensible to predictable interactions (10%).

timing management, side by side with the traditional VSync, to control the execution of new frames when animations or inputs update the screen content.

From the bottom up, when the screen HAL signals the HW-VSyncs, the traditional VSync module directly posts the software VSync events to the corresponding entities in the rendering pipeline at the configured frequency and offsets [24]. By contrast, *D-VSync* deploys the *Frame Pre-Executor* (§4.3) to explicitly control the timing and pacing of frame executions with D-VSync events posted prior to the display VSyncs when deterministic scenarios make pre-rendering feasible. Meanwhile, *D-VSync* configures larger buffer queues for frames to accumulate, recycling originally idle CPU time in short frames for the key (and long) frames to use. The screen can consume pre-rendered short frames accumulated in the buffer queue to avoid frame drops and reduce latency when a long frame emerges due to workload fluctuations.

The (D-)VSync events triggering the app UI thread and the render service/thread are equipped with timestamps for the OS UI framework and rendering logic to compute the frame content. *D-VSync* introduces the *Display Time Virtualizer* (§4.4) to decouple the conventional VSync timestamp of the current code execution time with a virtual displaying time of the frame, ensuring the correctness of frame content as if in the VSync architecture.

In addition, *D-VSync* includes a runtime controller and APIs (§4.5) for custom-rendering apps that bypass the data path provided by the OS rendering frameworks. Moreover, for decoupling-aware apps, *D-VSync* offers an extension of *Input Prediction Layer* (§4.6) to expand to the interactive frames.

4.2 The Scope of *D-VSync*

An important question is that *whether the decoupling and pre-rendering is feasible for most rendering cases* — so we are not proposing a technique for rare cases but for common cases. To answer the question, we further investigate the characteristics of all the frames for a typical user, as shown in Figure 9. Overall, frames to be rendered can be roughly classified into two categories: *animations* and *interactions*, while frames of animations are mostly deterministic.

We highlight three results. First, the majority of the frames in real-world are animations following a clicking operation. These include app opening, page transition, content loading, screen rotation, notification clearing, etc. All of these scenarios have noticeable impact on the user experience of a smartphone OS. *D-VSync* can enhance the smoothness of these deterministic pre-renderable animations, without requiring any effort from the upper-layer developers. Second, the yellow portions of the frames involve a real-time fingertip on the screen to trigger the movement that is usually predictable. These simple interactions, such as zooming or browsing on maps, require custom curve fitting of the input status to close the gap between the decoupled pre-rendering and displaying, which are categorized as *D-VSync* extensible. Last, for renderings that rely on sensor data or real-time online data (e.g., camera, or PvP games), *D-VSync* is not theoretically applicable. Yet, this should not be confused with the scene animations or UI transitions widely used in games.

As a result, our decoupled pre-rendering of *D-VSync* applies to all deterministic frames for animations (85%) and extends to simple interactive frames (10%), **covering 95% of the total frames in smartphones**. For complex real-time cases like online games or camera, *D-VSync* remains off.

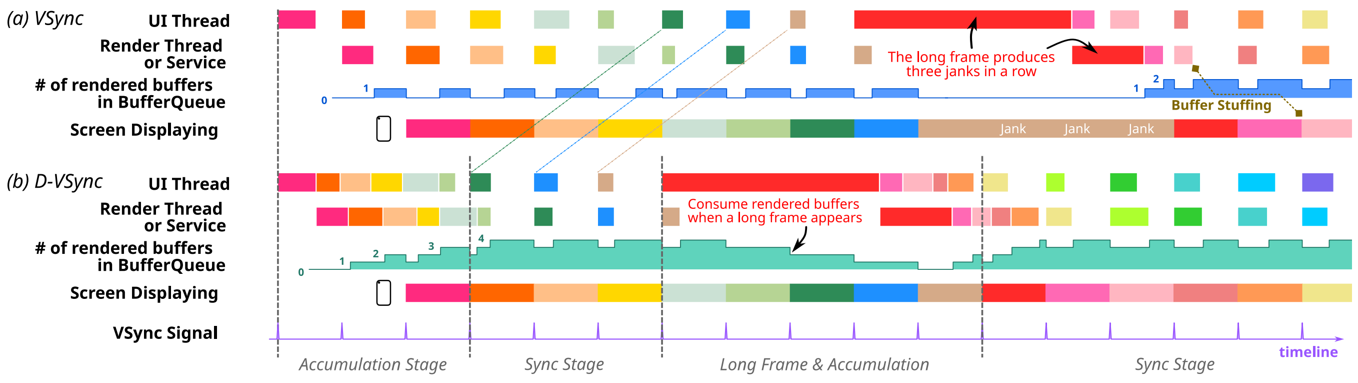

(a) VSync
UI Thread
Render Thread or Service
# of rendered buffers in BufferQueue
Screen Displaying
The long frame produces three janks in a row
Buffer Stuffing
Jank
(b) D-VSync
Consume rendered buffers when a long frame appears
VSync Signal
Accumulation Stage
Sync Stage
Long Frame & Accumulation
timeline

Figure 10. Execution patterns in VSync and *D-VSync*. The workloads of the same frame are dyed the same color. *D-VSync* has *the accumulation stage* and *the sync stage*. A long frame lets *D-VSync* consume rendered buffers to hide janks.

4.3 Frame Pre-Executor (*FPE*)

Frame Pre-Executor (*FPE*) performs decoupled pre-rendering to accumulate frames in deterministic scenarios. In the first step, *FPE* receives requests to execute the next frame in the last frame of the OS UI framework or the render service/thread. Such requests are generated when animations or inputs need to update the screen content. *D-VSync* modifies the UI framework to equip the requests with tags for all animations and simple interactions when pre-rendering is feasible. These requests cover more than *85%* of the total frames rendered in the smartphone (§4.2).

To pre-render frames in the decoupled manner, *FPE* posts D-VSync events ahead of the screen display VSyncs. When the last frame sending the request finishes, the next frame can get started in the corresponding rendering pipeline stage with the D-VSync event received. Figure 10 compares the runtime traces between VSync and *D-VSync*. In summary, *FPE* divides the rendering execution into 2 stages:

Accumulation Stage. In the original VSync architecture, frame logic is only triggered by the VSync signal under a specific frequency, as shown in Figure 10 (a). The buffer producer (software render thread/service) and the buffer consumer (hardware screen) both execute at the same rate enforced by the architecture, where a long frame produces three janks in a row. In contrast, *D-VSync* proactively computes frames in the beginning, as long as the pre-rendering does not exceed a configured threshold, i.e., there are still empty slots available in the buffer queue.

As illustrated in Figure 10 (b), because most short frames do not occupy the entire VSync period, the buffers can quickly get accumulated in the buffer queue as the screen slowly and steadily consumes them every VSync period. In this runtime trace, *D-VSync* configures 5 buffers in the buffer queue (1 front buffer for displaying with 4 back buffers for rendering) with the pre-rendering limit of 3 VSync periods. It is *the accumulation stage* that provides a foundation for the decoupled pre-rendering to function in *D-VSync*.

Sync Stage. *D-VSync* enters *the sync stage* as soon as the pre-execution reaches the limit where there is no remaining slot in the buffer queue. *FPE* triggers the execution of every frame in alignment with the screen display, similar to the conventional VSync architecture. The buffer consumption and the buffer production are in the same pacing, marked by the parallel dotted lines in Figure 10. The 2-stage design of *D-VSync* answers how and why frame drops can be reduced. For the exactly same series of workloads in VSync and *D-VSync*, the former produces three janks in a row while the latter is perfectly smooth. The screen consumes pre-rendered buffers rather than being unresponsive when a heavy key frame (the red frame in Figure 10) runs longer than expected. *D-VSync* thus presents a bigger time window to tolerate fluctuations.

4.4 Display Time Virtualizer (*DTV*)

Display Time Virtualizer (*DTV*) provides the new abstraction of frame display time to the UI and rendering framework, ensuring the content correctness during decoupled pre-rendering.

VSync architecture relies on the frame execution time to render the content, limiting the frame to be executed just shortly before it is displayed. Otherwise, the content will be outdated and there will be a noticeable lag. To break the close coupling between displaying and rendering, *DTV* brings *Frame Display Timestamp* (D-Timestamp in Figure 8), replacing the current wall clock or the VSync timestamp for frames to render their content. Animations use the D-Timestamp to sample motion curves for list flinging, app opening, page transition, screen rotation, etc. *DTV* guarantees that animations never appear fast in accumulation or slow down in long frames, with a uniform pacing just as the fixed VSync rhythm in the traditional architectures.

DTV computes when the next frame that *FPE* is going to trigger will be physically displayed on the screen. It is observed that the behaviors of the rendering system are deterministic: the screen HAL steadily consumes the buffer queue in FIFO order every VSync period, and the number of buffers queued and other configuration parameters such as the VSync period or offsets are always available to query. Based on the current states of the rendering system, *DTV* calculates the frame display timestamp and sends it to the UI and rendering framework with the D-VSync event.

As a result, applying *DTV*, *D-VSync* can pre-render frames while also ensuring correctness. Even if frames get rendered several VSync periods before, they are not aware of the time discrepancy for their content, as they can directly foresee when they will be displayed. *DTV* circumvents the latency issue due to waiting inside the buffer queue, and reduces the lengthened rendering latency after janks (in VSync).

4.5 Dual-channel Decoupling APIs

The compatibility with diverse apps is crucial to *D-VSync* for large-scale deployment on commercial smartphones. Hence, *D-VSync* proposes dual-channel decoupling APIs: it brings default improvements directly to decoupling-oblivious app binaries, as well as specialized enhancements for *decoupling-aware* apps modified in source code.

For existing OS native apps, the OS UI framework handles the cooperation with *D-VSync*, activating the decoupled pre-rendering in deterministic animation scenarios. The apps remain decoupling-oblivious, and runs correctly in both VSync and *D-VSync*-based systems (with better performance). No source code modification is needed.

For interactive scenarios when pre-rendering is not intuitive, or custom-rendering apps that use third-party rendering engines, the runtime controller turns off *D-VSync*, and the frame timing management defaults to the traditional VSync path. These cases, occupying less than 15% of the frames according to our characteristic study (§4.2), require *decoupling-aware APIs* to apply *D-VSync*. Specifically, *D-VSync* exposes the following capabilities for decoupling-aware pre-rendering: (1) an extensible Input Prediction Layer (§4.6), for interactive scenarios when the fingertip is physically on the screen; (2) configuration of the pre-rendering limit, which balances the performance and memory usage; (3) retrieval of the frame display time, for custom app-defined animations that bypass the OS UI framework; and (4) a runtime switch between *D-VSync* and VSync. §6.5 and §6.6 provide case studies to demonstrate how custom-rendering interactive apps can take advantage of *D-VSync*.

4.6 Extension: Input Prediction Layer (*IPL*)

D-VSync provides an extension of *Input Prediction Layer* (*IPL*) for decoupled pre-rendering in interactive frames when a fingertip (or a pen) is physically and continuously on the screen, as shown in Figure 8. These frames include zooming in a map, browsing an image/pdf, touching on a list for reading, etc. During the process of continuous user interactions, because *D-VSync* may execute frames several VSync periods before display, the latest status of the input events (e.g., touching coordinates) lying in between the rendering and displaying is missing. *IPL* aims to solve the issue.

IPL applies curve fitting to correct the current status of input events to the anticipated status at the expected frame display time computed by *DTV*. *IPL* is only activated for frames when the fingertip is physically on the screen, during a stream of inputs. For heavily-loaded scenarios, apps can register their specific heuristic curves through implementing an *IPL* interface. For instance, a map application can register linear line fitting when the users operate zooming, as rendering new vector tiles causes frame drops. *IPL* incurs minor overheads in practice as simple heuristic curves can fit the input patterns with very smooth user experience.

5 Implementation

We have implemented *D-VSync* in OpenHarmony 4.0 [23] and Android Open Source Project 13 (AOSP) [13]. *D-VSync* key modules take around 3,000 LOC in C++, and the peripheral code for modified components takes about 200 LOC.

5.1 *D-VSync* in OpenHarmony

We implement *D-VSync* on OpenHarmony with Mate40/60.
FPE. In OpenHarmony, the decoupled pre-execution pattern is applied in the Render Service (RS [18]), the OS rendering service, and the OS UI framework (ArkUI [7]). Every frame of the RS and the app UI thread is triggered by the decoupling-enhanced VSync-rs and VSync-app events, respectively. RS and ArkUI mark deterministic animations in their rendering and UI logic where decoupled pre-rendering is feasible. If it is the case, *FPE* triggers their pre-execution to accumulate short frames for later possible long frames. *FPE* keeps the state of pre-execution, i.e., the accumulation stage or the sync stage, allowing at most 3 back buffers for pre-rendering which tolerates most workload fluctuations in the wild.
DTV. The Display Time Virtualizer keeps a virtual clock of frame displaying based on the real hardware VSync signals as the frames accumulate. *DTV* maintains a model of the VSync timestamps, periods, and offsets. It calculates the expected Frame Display Timestamp (D-Timestamp) for the next frame triggered by the *FPE*, based on the current status of the rendering pipeline. *DTV* calibrates the issued D-Timestamp every few frames with hardware VSync signals to avoid error accumulation. *DTV* is also elastic to frame drops (still possible in *D-VSync*) and skips VSync periods in such cases.
API. The decoupling APIs expose the runtime capabilities of *D-VSync*. For interactive frames and custom-rendering apps, *D-VSync* accepts requests for the D-VSync signals that these scenarios can leverage for decoupled pre-rendering. The pre-rendering limit and the frame display timestamp computed are configurable and retrievable.

5.2 *D-VSync* in Android

We implement *D-VSync* on Android with Google Pixel 5.
FPE. In Android, the UI thread and the render thread of every app are triggered by the choreographer [17] listening on the VSync-app signals. *FPE* integrated inside the choreographer

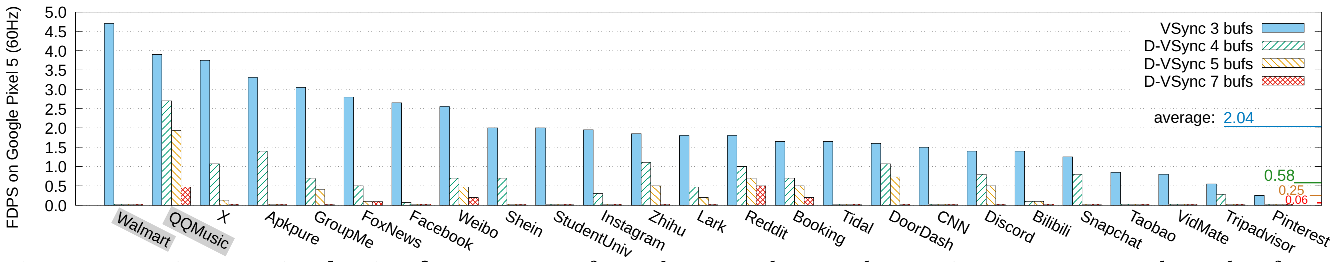


Figure 11. *D-VSync* FDPS reduction for apps. More frame drops are eliminated as *D-VSync* more proactively renders frames.

Table 1. Platform configuration

device	Pixel 5	Mate 40 Pro	Mate 60 Pro
release	Oct 2020	Nov 2020	Aug 2023
OS	AOSP 13	OH 4.0	OH 4.0
backend	GLES	GLES	GLES/VK
screen	1080 x 2340	1344 x 2772	1260 x 2720
refresh rate	60Hz / 16.7ms	90Hz / 11.1ms	120Hz / 8.3ms

keeps the *D-VSync* states and renders the next frame from the looper [21] when a pre-rendering decision is made.

DTV. The *DTV* collaborates with *FPE* to ensure that, the D-Timestamp always keeps constant pacing with the uniform screen refresh maintained by SurfaceFlinger [9], through necessary modelings and calibrations.

API. In addition to the APIs described in §5.1, *D-VSync* wires the *IPL* to the root of the view tree which delivers the input events. Apps can register custom curves for specific interactive scenarios using the exposed APIs.

5.3 *D-VSync* with LTPO

LTPO (low-temperature polycrystalline oxide) screens [31] support variable refresh rates. *D-VSync* aims to improve performance, and LTPO targets at lowering power consumption. Their benefits are mostly orthogonal from two dimensions.

Traditional LTPO approaches adjust the VSync frequency based on specific scenarios: a fixed 30 Hz for movies, 60 Hz for videos, and 120 Hz for interactions. *D-VSync* can adapt to such changes of refresh rates by adjusting the D-Timestamp calculation based on the scenarios as well.

State-of-the-art LTPO approaches, such as Apple's ProMotion [12], Huawei's X-True [19], and Oppo's O-Sync [14], dynamically lower the VSync frequency in real time when the motion of animations is slow enough for human eyes to perceive the difference. For example, a swipe may start at 120 Hz, and when the scrolling speed of the list drops below a certain threshold, the refresh rate switches to 90 Hz and eventually to 60 Hz. To ensure consistent rendering and displaying (frames rendered at X Hz are not displayed at Y Hz), *D-VSync* cooperates with the LTPO module to coordinate the timing of rendering rate change and refresh rate change in terms of accumulated frames if any. The frames produced at frame rate X must be consumed by the screen's HAL before the screen can switch to the new refresh rate Y. Every rendered buffer has a rendering rate bound to it, controlling the display duration of the frame, as well as the timing of the screen refresh rate change when two adjacent frames have different rates.

The co-design (with LTPO) has also been implemented in commercial smartphone OSes and devices.

6 Evaluation

We evaluate *D-VSync* on Google Pixel 5 (60 Hz) with AOSP 13, and Mate 40 Pro (90 Hz) and Mate 60 Pro (120 Hz) with OpenHarmony 4.0. Table 1 illustrates the device configurations. We report both **the objective data** of *frame drops* collected by industrial testing framework, and **the subjective data** of *stutters* assessed by professional user experience (UX) evaluators. We also evaluate the *rendering latency*, *execution cost*, *power consumption*, and present *case studies* of decoupling-aware apps (map and Chromium [3]).

6.1 Reduction of Frame Drops

We collect the objective data through a testing framework from a major smartphone vendor. It simulates necessary real-user clicks and swipes for a defined set of scenarios. Specifically, we report data from 25 world top apps covering common categories, 75 OS use cases and apps encompassing most system functionalities[1], and simulations of *D-VSync* on 15 mobile games. These testings embody years of experiences from the smartphone deployment, represent real user behaviors, and provide a reference for future research.

We first evaluate the performance of frame drop reduction under different configurations of pre-rendering limit. As *D-VSync* more proactively accumulates frames, frame drops per second (FDPS) are reduced more, while the memory cost for buffers will be higher (§6.4). As illustrated in Figure 11, for each app and each configuration, 1000 frames in Google Pixel 5 are recorded by swiping the main page twice a second. When *D-VSync* is allowed to utilize 4 buffers, the default setting of *D-VSync*, 71.6% of frame drops are eliminated, and FDPS averages at 0.58. For 5 buffers, FDPS reduction reaches 87.7% and only one jank is detected for every 4 seconds.

To further prove the effectiveness of *D-VSync*, we evaluate 75 common OS use cases and apps on Mate 40 Pro and Mate 60 Pro that have frame drops in VSync. Averages are derived from five runs to mitigate fluctuations. For the 9

[1] Detailed descriptions and abbreviations are listed in Appendix A.

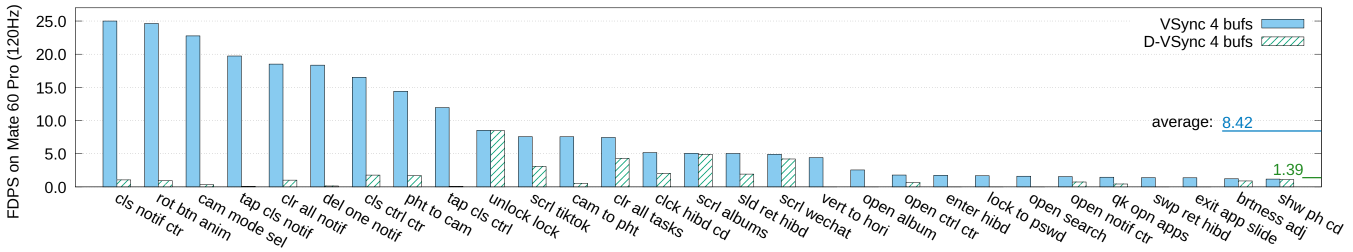
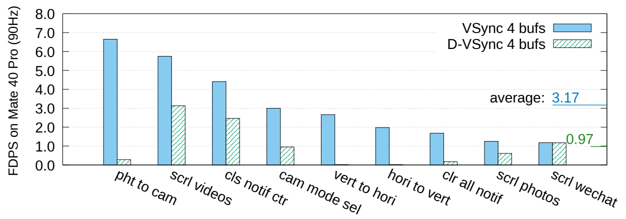


Figure 12. *D-VSync* FDPS reduction for common OS use cases with Vulkan backend. Appendix A lists the abbreviations.



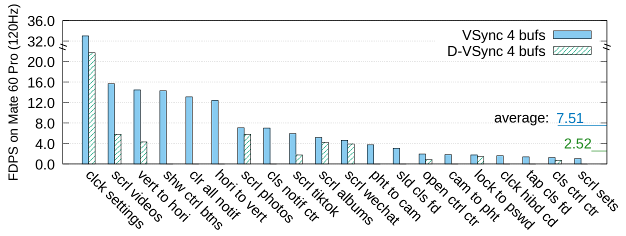


Figure 13. *D-VSync* FDPS reduction with GLES.

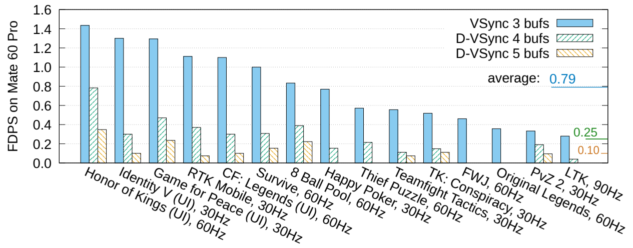


Figure 14. Simulation in games for FD reduction.

cases that have frame drops on Mate 40 Pro, the average FDPS is reduced from 3.17 to 0.97 by 69.4%, as shown in Figure 13. For Mate 60 Pro, the reduction is 83.5% and 66.4% for Vulkan and GLES backends respectively, illustrated in Figure 12 and 13. Overall, the average frame drop reduction on the three devices with *D-VSync* equipped is 72.7%.

Analysis. We compare Walmart and QQMusic in Figure 11, where *D-VSync* significantly improves the former, not the latter. For Walmart, frame drops are scattered in time and most long frames take less than 3 VSync periods, which is effectively addressed by the decoupled pre-rendering of *D-VSync*. In contrast, QQMusic has a considerably skewed distribution, with numerous long frames that even 7 buffers fail to hide the jank. Therefore, *D-VSync* is not a panacea, and inefficient code is still discouraged while *D-VSync* can reduce frame drops to a large extent.

Simulation of Games. We collect the runtime traces (CPU and GPU time of every frame) of 15 mobile games for UI and scene animations where frame drops are common. We then use scripts to simulate the *D-VSync* decoupled pre-rendering pattern and count frame drops. These games use custom rendering engines that bypass the *D-VSync* framework. Figure 14 illustrates the theoretical performance boosts of 68.4% and 87.3% FDPS reduction using 4 or 5 buffers, respectively. We are working with these third-party partners to utilize the decoupling-aware APIs.

Table 2. Number of stutters preceived in professional user experience (UX) evalution of different tasks.

Task Description	VSync	*D-VSync*
Cold start and close the Top 20 apps, then slide through the multitasking interface.	20	12 ↓40%
Cold start every Top 10 news/social apps, and immediately swipe upwards after start.	28	3 ↓89%
Hot start every Top 10 news/social apps, and immediately swipe upwards after start.	25	2 ↓92%
In a game app, switch to a news app and swipe upwards (switch back to the game and repeat 5 times)	20	3 ↓85%
In a short video app, open up the comments and swipe upwards (slide to the next video and repeat 5 times)	20	2 ↓90%
In a music app, swipe through the music page and click on one to play (switch back and repeat 5 times)	7	0 ↓100%
In a shopping app, swipe through the products page, and open up a product to swipe through the details.	14	13 ↓7%
In a lifestyle app, swipe through the advertisements, and open up all nearby restaurants to swipe through.	40	10 ↓75%

6.2 Reduction of User-perceived Stutters

We collect the subjective data assessed by our professional user experience (UX) team. The UX evaluators are well trained to perform the tasks listed in Table 2 and report any stutters they perceive. The janks are later confirmed by a high-speed camera recording the screen. Compared to the objective testing cases, every task of UX evaluation involves multiple consecutive operations in different scenes and pages. Table 2 compares the number of stutters perceived in VSync and *D-VSync* on Mate 60 Pro. *D-VSync* on average reduces 72.3% of the stutters over all the tasks, improving the user experience.

6.3 Reduction of Rendering Latency

D-VSync circumvents the unnecessary waiting latency caused by buffer stuffing in the triple-buffering VSync architecture. Through the decoupled Display-Timestamp, frames in *D-VSync* are able to proactively accumulate with a future timestamp at hand to avoid any time discrepancy.

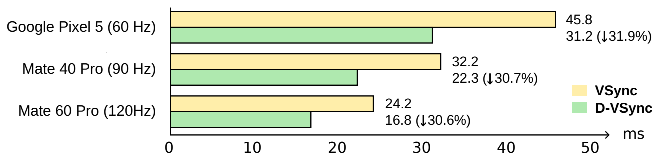


Figure 15. Rendering latency reduction.

To evaluate the rendering latency, we write a script to measure the duration from the execution of a frame (or the D-Timestamp in *D-VSync*) to its final display (i.e., present fence indicated by the screen HAL), for buffer-stuffing frames, direct-composition (i.e., no waiting) frames, and frame drops, across all the runtime traces we collect.

Figure 15 illustrates a summary on the three devices with different refresh rates. On average, *D-VSync* reduces the rendering latency by 31.1%.

6.4 Costs of *D-VSync*

Execution time. The pre-execution management and the display time computation create *minor execution overheads*. The *D-VSync* module spends an additional 102.6 µs of execution time every frame compared with the VSync baseline, including both the *FPE* and *DTV*. The cost is negligible, occupying 1.2% of a 120 Hz VSync period. Besides, VSync/*D-VSync* threads run on little cores that do not compete with the UI/rendering threads on middle/big cores. §6.5 evaluates the cost from the app side using decoupling-aware APIs.

Memory. *D-VSync* has *more memory usage* as we enlarge the maximum number of frame buffers. A full-screen RGBA8888 frame buffer takes around 10 MB in Pixel 5 and 15 MB in Mate phones. In Android, *D-VSync* that utilizes 4 buffers takes about 10 MB per app in addition to the triple-buffering VSync, permissible in modern smartphones with at least 8 GB of memory. For Mate 40 Pro and Mate 60 Pro, since the render service by default uses 4 buffers for redundancy, *D-VSync* does not cause any noticeable increase in memory usage, with less than 10 KB used for the *FPE*, *DTV*, and API logic itself.

6.5 Case Study-1: Map App

We demonstrate the potential of *D-VSync* for decoupling-aware zoom operations by developing a map app using the AMap SDK [1] on Android. Zooming requires two fingers to remain on the screen for real-time adjustment of the zoom level. During this process, different levels of vector tiles (geographic data) are loaded and rendered. Compared with browsing, zooming has a heavier load with frame drops.

To reduce stutters, the map app registers a Zooming Distance Predictor (*ZDP*) using the *IPL* extension in *D-VSync*. Users do zooming by increasing or decreasing the distance between the two fingertips. *ZDP* outputs the expected distance at the D-Timestamp retrieved from the *DTV* API, based on a linear line fitting of current (and historical) data of the distance. Through the runtime controller API, *D-VSync* is only

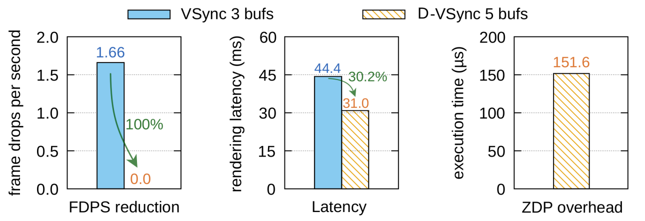


Figure 16. Evaluation of the map app. Custom-rendering interactive apps can also take advantage of *D-VSync*.

activated in zooming, not browsing. For the pre-rendering limit, the map configures *D-VSync* to utilize 5 buffers.

The additional logic takes less than 200 LOC in Java, while it eliminates 100% of frame drops and enhances the smoothness significantly. For 3,600 frames recorded, the latency is reduced by 30.2%, with an average execution time of 151.6 µs every frame for *ZDP*, shown in Figure 16.

6.6 Case Study-2: Chromium Browser

Chromium [3] is an open-source web browser whose rendering pipeline takes HTML plain texts with data and scripts to produce pixel bitmaps as frames (i.e., a custom-rendering app). A web page is divided into layers with tiles that are rasterized asynchronously and later composited synchronously with VSync signals. We developed the decoupled rendering scheme on the real-time compositor of Chromium in OpenHarmony, which pre-renders frames during animations using *D-VSync*-aware APIs. For Sina, Weather, and AI Life pages, the average FDPS is reduced from 1.47 to 0.08 by 94.3% during the flinging animations after swiping.

6.7 Power Consumption

The additional power consumption is essentially negligible, as *D-VSync* merely shifts subsequent loads forward, allowing more frames to be completed before the VSync deadline. Specifically, the end-to-end power consumption of the device increases 0.13%–0.37%. This overhead includes the rendering of frames that would have been dropped in the VSync architecture (theoretically should have been rendered).

End-to-end power consumption. We use a power tester connected to a fully charged Pixel 5, tracking the power for 30 minutes after a 30-minute warmup. For an animation programmed in the map app (both in VSync and in *D-VSync* using *FPE*, *DTV* and APIs), the power consumption increases 0.13%. When 10% of the frames additionally invoke the *ZDP* input curve fitting, the power increase reaches 0.37%.

CPU instructions. For the 75 OS use cases on Mate 60 Pro, we record the number of CPU instructions executed in the render service. With *D-VSync* on and off, the averages are 10.849 and 10.793 million instructions per frame. The additional overhead is 0.52%.

7 Deployment Experience

We derived three lessons from our deployment of *D-VSync*.

Subjective data are equally important, particularly for rendering systems interacting with end users. Some performance aspects elude objective metrics. For example, incorrect frame display time calculations can cause chaotic content despite achieving higher frame rates. A comprehensive user experience evaluation prior to deployment can provide a more accurate reflection of the system's benefits.

Educating app/system developers is crucial, especially for the deployment of a new architecture. Graphics programmers ofter rely on runtime traces to locate performance bottlenecks, while our new architecture fundamentally alters the execution pattern. Therefore, we held lectures within the company, exchanged ideas with top third-party application partners, and wrote detailed architecture documentation.

Compatibility should be considered in advance, when different related solutions are being developed in parallel. *D-VSync* and LTPO solutions were developed almost simultaneously on HarmonyOS NEXT by different teams. Later, we realized the potential interplay between the two, leading to significant time spent resolving conflicts and correcting designs. Early compatibility considerations and better communication between teams would have increased efficiency.

8 Related Work

State-of-the-art research regarding the rendering system usually focuses on optimizing one component in the whole architecture [26, 27, 29, 33, 38, 44, 47, 49–51, 55, 60], proposing new hardware extensions [45, 62–65, 68], or making use of other system components to aid the rendering pipeline, such as energy-aware scheduling, DVFS governing, or touch boosting [32, 34–36, 42, 48, 52, 67] However, they mainly target at energy consumption. Improvements for frame drops and latency are less focused or outstanding.

For optimizations relevant to *D-VSync* in mobile rendering, LTPO [12, 14, 19, 31] supports variable screen refresh rates to save power. Arnau et al. [28] propose parallel frame rendering where two consecutive frames are rendered in parallel to trade responsiveness for energy on mobile GPU, while *D-VSync* is a solution to hide the display latency. Pathania et al. [57, 58] implement an integrated CPU-GPU power management strategy to adjust FPS to a target lower than 60 to save energy. However, 50 FPS in smartphones without G-Sync [5] or FreeSync [2] implies that 10 janks are produced on a 60 Hz screen. Lo et al. [52] and Choi et al. [34] estimate the execution time of frames to adjust the CPU/GPU frequency, so each job can finish just before the VSync signal to save energy. We consider these works orthogonal but applicable to *D-VSync* which gives a bigger time window for frame execution.

Similar techniques of *D-VSync* has been used in various settings of mobile computing. Huang et al. [41], Yan et al. [66], and Baeza-Yates et al. [30] use contextual information to infer the next app to be used, facilitating app launching. Agrawal et al. [26] pre-inflate UI layouts to reduce activity transition time. PES [36] proactively anticipates events in the DOM of web apps to better coordinate scheduling. On the other hand, *D-VSync* pre-renders deterministic frames where screen refresh imposes a more strict millisecond-level deadline. Outatime [46] predicts navigation inputs to mask network latency in cloud gaming using a Markov model. Hou et al. [40] use LSTM and MLP to predict head and body motion in VR. It is possible to integrate their predictors into *D-VSync* for various interactive scenarios.

9 Conclusion

This paper introduces *D-VSync*, a new rendering architecture for smartphone OSes that eliminates frame drops and reduces latency. Based on the power law distribution of rendering workloads, *D-VSync* decouples the rendering execution from the periodic display events so that accidental long frames can use the computational power saved by common short frames. *D-VSync* has been deployed as a pivotal innovation in the next generation of major smartphone products. We anticipate that it will have a significant impact on the evolution of rendering architectures.

Acknowledgments

We sincerely thank our shepherd Yuhao Zhu and the anonymous ASPLOS'25 reviewers for their insightful suggestions. We sincerely thank Xiaobo Xu, Shaoquan Qin, Hui Guo, and other colleagues from Huawei Fields Lab for the commercial implementation and deployment of *D-VSync*. This work was supported in part by National Natural Science Foundation of China (No. 62132014, 61925206, 62472279, 62302300), and Startup Fund for Young Faculty at SJTU (SFYF at SJTU). Corresponding authors: Dong Du (dd_nirvana@sjtu.edu.cn) and Yubin Xia (xiayubin@sjtu.edu.cn).

A Appendix

A.1 Abstract

This appendix lists the 75 common OS use cases and apps that the paper uses to conduct end-to-end evaluations on the VSync-based and *D-VSync*-based systems. These cases are carefully selected to cover a wide range of system functionalities, such as unlocking the phone, rotating the screen, opening apps, etc., as well as a set of commonly used system apps, such as the camera, photos, control center, notification center, etc. These use cases form a benchmark that comprehensively tests the performance of the OS rendering service, providing a reference for the follow-up research.

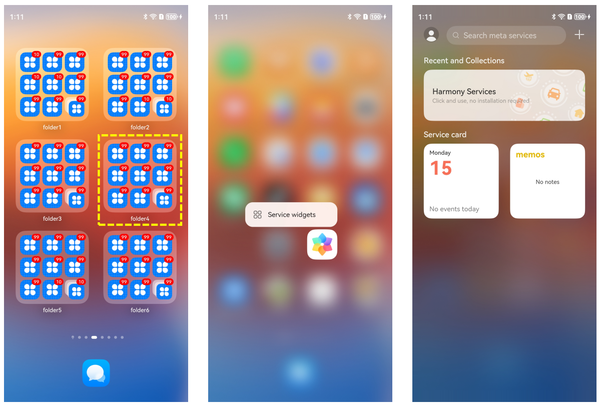
A.2 Methodology

The 75 OS use cases and apps are fully automated through python scripts that mimic necessary human operations. A phone is connected to the PC through a USB cable, and script commands are sent through OpenHarmony Device

Table 3. Detailed descriptions of the OS use cases and apps for end-to-end rendering performance evaluation.

	Category	Detailed Description	Abbreviation
1	Phone Unlocking	Swipe upwards in the lock screen to enter the password page	lock to pswd
2	Phone Unlocking	The fly-in animation of the sceneboard after entering the last digit of the password	pswd to desk
3	Phone Unlocking	Swipe upwards in the lock screen to unlock the phone (without password)	unlock lock
4	Phone Unlocking	The fly-in animation of the sceneboard (without password)	lock to desk
5	Sceneboard	Slide the sceneboard pages left and right (with default pre-installed apps)	slide desk
6	Sceneboard	Slide the sceneboard pages left and right when exiting an app	exit app slide
7	Sceneboard	Slide the sceneboard pages left and right with full folders (Figure 17(a))	slide full fd
8	App Operation	App opening animation when clicking an app	open app
9	App Operation	App closing animation when swiping upwards	close app
10	App Operation	App closing animation when sliding rightwards	sld cls app
11	App Operation	Quickly open and close apps one after another	qk opn apps
12	Folder	Folder opening animation when clicking a folder	open fd
13	Folder	Folder closing animation when tapping the empty space outside	tap cls fd
14	Folder	Folder closing animation when sliding rightwards	sld cls fd
15	Folder	Folder closing animation when swiping upwards	swp cls fd
16	Cards	Long click the photos app and the cards show up (Figure 17(b))	shw ph cd
17	Cards	Tap the empty space outside to close the cards of the photos app	cls ph cd
18	Cards	Long click the memos app and the cards show up	shw mem cd
19	Cards	Tap the empty space outside to close the cards of the memos app	cls mem cd
20	Notification Center	Swipe downwards to open the notification center	open notif ctr
21	Notification Center	Swipe upwards to close the notification center	cls notif ctr
22	Notification Center	Tap the empty space to close the notification center	tap cls notif
23	Notification Center	Click the trash can button to clear all notifications	clr all notif
24	Notification Center	Slide rightwards to delete one notification and the bottom ones move up	del one notif
25	Control Center	Swipe downwards to open the control center	open ctrl ctr
26	Control Center	Swipe upwards to close the control center	cls ctrl ctr
27	Control Center	Tap the empty space to close the control center	tap cls ctrl
28	Control Center	Click the unfold button to show all control buttons	shw ctrl btns
29	Control Center	Screen rotation button animation when clicking on the button	rot btn anim
30	Control Center	Click the settings button in the control center to enter the settings	clck settings
31	Control Center	Adjust the screen brightness in the control center	brtness adj
32	Volume Bar	The volume bar appears when clicking the physical volume adjustment button	shw vol bar
33	Volume Bar	Disappearing animation of the volume bar after some time of no operation	vol bar gone
34	Volume Bar	Short click the physical volume adjustment button to adjust volume	clck adj vol
35	Volume Bar	Long click the physical volume adjustment button to adjust volume	lclck adj vol
36	Volume Bar	Slide the volume bar on the screen to adjust volume	sld adj vol
37	Volume Bar	Tap the empty space to hide the volume bar	hide vol bar
38	Tasks	Swipe upwards on the sceneboard to enter tasks	opn tasks dsk
39	Tasks	Swipe upwards on the app to enter tasks	opn tasks app
40	Tasks	Slide the tasks left and right	sld tasks
41	Tasks	Swipe upwards to delete one task and the last task moves rightwards	del one task
42	Tasks	Click the trash can button to clear all tasks and go back to the sceneboard	clr all tasks
43	Tasks	Tap the empty space to leave the tasks	leave tasks
44	Tasks	Click one task to enter the app	task open app
45	HiBoard	Slide rightwards from the first page of the sceneboard to enter HiBoard (Figure 17(c))	enter hibd
46	HiBoard	Click the weather card on HiBoard to enter weather app	clck hibd cd
47	HiBoard	Swipe upwards in the weather app to return to HiBoard	swp ret hibd
48	HiBoard	Slide rightwards in the weather app to return to HiBoard	sld ret hibd
49	Global Search	Swipe downwards to open global search	open search
50	Global Search	Slide rightwards to close global search	cls search
51	Keyboard	Click the browser search bar to show the virtual keyboard	shw kb
52	Keyboard	Click the keyboard hide button to hide the virtual keyboard	hide kb
53	Screen Rotation	Rotate the screen from vertical to horizontal when displaying a full-screen photo	vert ph hori
54	Screen Rotation	Rotate the screen from horizontal to vertical when displaying a full-screen photo	hori ph vert

	Category	Detailed Description	Abbreviation
55	Screen Rotation	Rotate the screen from vertical to horizontal when displaying an app	vert to hori
56	Screen Rotation	Rotate the screen from horizontal to vertical when displaying an app	hori to vert
57	Photos	Scroll the albums in the photos app	scrl albums
58	Photos	Click into one album and enter its photo list	open album
59	Photos	Scroll the photo list in the photos app	scrl photos
60	Photos	Click into one photo and view the photo in full screen	clck photo
61	Photos	Browse the full-screen photo	brws photo
62	Photos	Swipe downwards the full-screen photo to return to the photo list	ret photos
63	Photos	Slide rightwards the full-screen photo to return to the photo list	sld ret photos
64	Photos	Click the back button in the photo list to return to the album list	ret albums
65	Camera	Click the photo preview in the camera app to enter the photos app	cam to pht
66	Camera	Slide rightwards from the photos app to return to the camera app	pht to cam
67	Camera	Slide inside the camera app to select between camera modes	cam mode sel
68	Browser	Click the pages button to see all the opening pages in the browser app	brwsr pages
69	Settings	Scroll the settings in the main page of the settings app	scrl sets
70	Settings	Click the bluetooth setting in the settings app to enter the subpage	clck bt
71	Settings	Click the WLAN setting in the settings app to enter the subpage	clck wlan
72	Settings	Click the login tab in the settings app to enter the subpage	clck login
73	Other Apps	Scroll the main page of WeChat	scrl wechat
74	Other Apps	Scroll the videos of TikTok	scrl tiktok
75	Other Apps	Scroll the video lists of Videos	scrl videos



(a) Full Folders **(b) App Cards** **(c) HiBoard**

Figure 17. (a) A sceneboard page full of folders, where the dashed box encompasses a folder. **(b)** App cards show up when long clicking an app. **(c)** HiBoard appears when sliding rightwards on the first page of the sceneboard.

Connector (HDC) [15], a tool similar to ADB [16] that debugs and controls the device. Before any test starts, a setup script installs required apps, positions them correctly on the sceneboard (desktop), and pushes dummy data like photos and videos onto the phone. Every test case starts and ends on the first page of the sceneboard. Each script first enters the start position of the test from the first page, and then records a trace while performing simulated clicks or swipes. Finally, it counts the number of frame drops and calculates the frame drops per second (FDPS) from the trace. Averages are derived from five runs to mitigate fluctuations.

A.3 Detailed Descriptions

Detailed descriptions of the use cases are shown in Table 3. The corresponding abbreviations are used in Figure 12 and 13 of the paper.

References

[1] Amap open platform | amap api. https://lbs.amap.com/.
[2] Amd freesync™ technology. https://www.amd.com/en/technologies/free-sync.
[3] Chromium. https://www.chromium.org/Home/.
[4] Compositor thread architecture. https://www.chromium.org/developers/design-documents/compositor-thread-architecture/.
[5] Nvidia g sync: The best gaming monitors. https://www.nvidia.com/en-us/geforce/products/g-sync-monitors/.
[6] Nvidia reflex victory measured in milliseconds. https://www.nvidia.com/en-us/geforce/technologies/reflex/.
[7] Openharmony arkui_ace_engine. https://gitee.com/openharmony/arkui_ace_engine.
[8] Perfetto - system profiling, app tracing and trace analysis - perfetto tracing docs. https://perfetto.dev/docs/.
[9] Surfaceflinger and windowmanager. https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager.
[10] Understanding user interface responsiveness. https://developer.apple.com/documentation/xcode/understanding-user-interface-responsiveness/.
[11] Opengl es - the standard for embedded accelerated 3d graphics, 07 2011. https://www.khronos.org/api/opengles.
[12] ipad pro, in 10.5-inch and 12.9-inch models, introduces the world's most advanced display and breakthrough performance, 06 2017. https://www.apple.com/newsroom/2017/06/ipad-pro-10-5-and-12-9-inch-models-introduces-worlds-most-advanced-display-breakthrough-performance/.
[13] Android open source project, 2019. https://source.android.com/.
[14] Oppo find x3 pro | oppo global, 03 2021. https://www.oppo.com/en/smartphones/series-find-x/find-x3-pro/.

[15] hdc, 12 2022. https://docs.openharmony.cn/pages/v4.0/en/device-dev/subsystems/subsys-toolchain-hdc-guide.md/.

[16] Android debug bridge (adb), 08 2023. https://source.android.com/docs/setup/build/adb.

[17] Choreographer, 04 2023. https://developer.android.com/reference/android/view/Choreographer.

[18] Graphics subsystem, 04 2023. https://gitee.com/openharmony/docs/blob/master/en/readme/graphics.md.

[19] Huawei mate x3 - huawei global, 03 2023. https://consumer.huawei.com/en/phones/mate-x3/.

[20] ios 17, 12 2023. https://www.apple.com/ios/ios-17/.

[21] Looper, 02 2023. https://developer.android.com/reference/android/os/Looper.

[22] Nativevsync, 12 2023. https://docs.openharmony.cn/pages/v4.0/en/application-dev/reference/native-apis/_native_vsync.md/.

[23] Openatom openharmony, 12 2023. https://www.openharmony.cn/mainPlay.

[24] Vsync, 11 2023. https://source.android.com/docs/core/graphics/implement-vsync.

[25] Vulkan | cross platform 3d graphics, 2024. https://www.vulkan.org/.

[26] Sumeen Agrawal, Manith Shetty, Sripurna Mutalik, and Anuradha Kanukotla. Method to improve ui rendering using predictive sequence modelling. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 5031–5037, 2022. https://doi.org/10.1109/ICPR56361.2022.9956234.

[27] Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan L. Aragón, Pedro Marcuello, and Antonio González. Rendering elimination: Early discard of redundant tiles in the graphics pipeline. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 623–634, 2019. https://doi.org/10.1109/HPCA.2019.00014.

[28] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 83–92, 2013. https://doi.org/10.1109/PACT.2013.6618806.

[29] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 529–540, 2014. https://doi.org/10.1109/ISCA.2014.6853207.

[30] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. Predicting the next app that you are going to use. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, page 285–294, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2684822.2685302.

[31] Ting-Kuo Chang, Chin-Wei Lin, and Shihchang Chang. 39-3: Invited paper: Ltpo tft technology for amoleds†. *SID Symposium Digest of Technical Papers*, 50(1):545–548, 2019. https://doi.org/10.1002/sdtp.12978.

[32] Wei-Ming Chen, Sheng-Wei Cheng, Pi-Cheng Hsiu, and Tei-Wei Kuo. A user-centric cpu-gpu governing framework for 3d games on mobile devices. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 224–231, 2015. https://doi.org/10.1109/ICCAD.2015.7372574.

[33] Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. Tree traversal synthesis using domain-specific symbolic compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1030–1042, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3503222.3507751.

[34] Yonghun Choi, Seonghoon Park, and Hojung Cha. Graphics-aware power governing for mobile devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 469–481, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3307334.3326075.

[35] Yonghun Choi, Seonghoon Park, Seunghyeok Jeon, Rhan Ha, and Hojung Cha. Optimizing energy consumption of mobile games. *IEEE Transactions on Mobile Computing*, 21(10):3744–3756, 2022. https://doi.org/10.1109/TMC.2021.3058381.

[36] Yu Feng and Yuhao Zhu. Pes: Proactive event scheduling for responsive and energy-efficient mobile web computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 66–78, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3307650.3322248.

[37] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity: High-Performance mobile emulation through graphics projection. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 285–301, Carlsbad, CA, July 2022. USENIX Association. https://www.usenix.org/conference/osdi22/presentation/gao.

[38] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. Every pixel counts: Fine-grained ui rendering analysis for mobile applications. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017. https://doi.org/10.1109/INFOCOM.2017.8057023.

[39] Chet Haase and Romain Guy. For butter or worse: Smoothing out performance in android uis, 01 2012. https://www.youtube.com/watch?v=Q8m9sHdyXnE.

[40] Xueshi Hou and Sujit Dey. Motion prediction and pre-rendering at the edge to enable ultra-low latency mobile 6dof experiences. *IEEE Open Journal of the Communications Society*, 1:1674–1690, 01 2020. https://doi.org/10.1109/OJCOMS.2020.3032608.

[41] Ke Huang, Chunhui Zhang, Xiaoxiao Ma, and Guanling Chen. Predicting mobile application usage using contextual information. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, page 1059–1065, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2370216.2370442.

[42] Nohyun Jung, Gwangmin Lee, Seokjun Lee, and Hojung Cha. Tbooster: Adaptive touch boosting for mobile texting. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*, HotMobile '16, page 63–68, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2873587.2873592.

[43] Kaylin. Phone screen time addiction- new survey data & statistics, 01 2024. https://www.harmonyhit.com/phone-screen-time-statistics/.

[44] Gwangmin Lee, Seokjun Lee, Geonju Kim, Yonghun Choi, Rhan Ha, and Hojung Cha. Improving energy efficiency of android devices by preventing redundant frame generation. *IEEE Transactions on Mobile Computing*, 18(4):871–884, 2019. https://doi.org/10.1109/TMC.2018.2844202.

[45] Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong Sim. Gscore: Efficient radiance field rendering via architectural support for 3d gaussian splatting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 497–511, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3620666.3651385.

[46] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 151–165, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2742647.2742656.

[47] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoxing Ma, and Jian Lü. Characterizing and detecting inefficient image displaying issues in

android apps. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 355–365, 2019. https://doi.org/10.1109/SANER.2019.8668030.

[48] Xianfeng Li and Gengchao Li. An adaptive cpu-gpu governing framework for mobile games on big.little architectures. *IEEE Transactions on Computers*, 70(9):1472–1483, 2021. https://doi.org/10.1109/TC.2020.3012987.

[49] Xianfeng Li and Gengchao Li. Hb-retriple: Mobile rendering optimization based on efficient history reusing. *Journal of Systems Architecture*, 129:102627, 2022. https://doi.org/10.1016/j.sysarc.2022.102627.

[50] Xianfeng Li, Gengchao Li, and Xiaole Cui. Retriple: Reduction of redundant rendering on android devices for performance and energy optimizations. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. https://doi.org/10.1109/DAC18072.2020.9218517.

[51] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. Conflict-driven synthesis for layout engines. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. https://doi.org/10.1145/3591246.

[52] Daniel Lo, Taejoon Song, and G. Edward Suh. Prediction-guided performance-energy trade-off for interactive applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 508–520, 2015. https://doi.org/10.1145/2830772.2830776.

[53] Katerina Mania, Bernard D. Adelstein, Stephen R. Ellis, and Michael I. Hill. Perceptual sensitivity to head tracking latency in virtual environments with varying degrees of scene complexity. In *Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization*, APGV '04, page 39–47, New York, NY, USA, 2004. Association for Computing Machinery. https://doi.org/10.1145/1012551.1012559.

[54] Ann McNamara, Katerina Mania, and Diego Gutierrez. Perception in graphics, visualization, virtual environments and animation. In *SIGGRAPH Asia 2011 Courses*, SA '11, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2077434.2077448.

[55] Jiayi Meng, Sibendu Paul, and Y. Charlie Hu. Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 923–937, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378516.

[56] Khalid Moammer. Amd freesync vs nvidia g-sync - dissected and compared, 03 2015. https://wccftech.com/amd-freesync-nvidia-gsync-verdict/.

[57] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2744769.2744894.

[58] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2593069.2593151.

[59] Rob Shafer. What is screen tearing and how do you fix it? [simple guide], 03 2022. https://www.displayninja.com/what-is-screen-tearing/.

[60] Wei Song, Mengqi Han, and Jeff Huang. Imgdroid: Detecting image loading defects in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 823–834, 2021. https://doi.org/10.1109/ICSE43902.2021.00080.

[61] Josef Spjut, Ben Boudaoud, Kamran Binaee, Jonghyun Kim, Alexander Majercik, Morgan McGuire, David Luebke, and Joohwan Kim. Latency of 30 ms benefits first person targeting tasks more than refresh rate above 60 hz. In *SIGGRAPH Asia 2019 Technical Briefs*, SA '19, page 110–113, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3355088.3365170.

[62] Blaise Tine, Varun Saxena, Santosh Srivatsan, Joshua R. Simpson, Fadi Alzammar, Liam Cooper, and Hyesoon Kim. Skybox: Open-source graphic rendering on programmable risc-v gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 616–630, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3582016.3582024.

[63] Nisarg Ujjainkar, Jingwen Leng, and Yuhao Zhu. Imagen: A general framework for generating memory- and power-efficient image processing accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3579371.3589076.

[64] Nisarg Ujjainkar, Ethan Shahan, Kenneth Chen, Budmonde Duinkharjav, Qi Sun, and Yuhao Zhu. Exploiting human color discrimination for memory- and energy-efficient image encoding in virtual reality. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 166–180, New York, NY, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3617232.3624860.

[65] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-vr: system-level design for future mobile collaborative virtual reality. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 587–599, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3445814.3446715.

[66] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 113–126, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2307636.2307648.

[67] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, 2015. https://doi.org/10.1109/HPCA.2015.7056028.

[68] Yuhao Zhu and Vijay Janapa Reddi. Webcore: Architectural support for mobile web browsing. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 541–552, 2014. https://doi.org/10.1109/ISCA.2014.6853239.