Phantom Rendering Detection: Identifying and Analyzing Unnecessary UI Computations

ZHIHAO LIN*, Beihang University, China
MINGYI ZHOU*, Beihang University, China
BO SUN, Huawei, China
HAN HU, Huawei Hong Kong Research Center, Hong Kong
GANG FAN, Huawei Hong Kong Research Center, Hong Kong
LI LI†, Beihang University, China

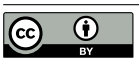
Modern mobile applications have high-resolution user interfaces (UI) and heavy computations, resulting in significant energy consumption and latency, especially on older devices. Precisely measuring the performance of mobile operations (*e.g.*, the number of CPU instructions) and detecting performance issues are critical for mobile software engineering. In this study, we characterize a previously underexplored class of performance issues on mobile called *Phantom Rendering*, which occurs when mobile applications perform unnecessary UI-related offscreen computations but do not visually render them. For example, the animation component stops visually rendering on the screen but continues to refresh in the background. This problem represents a root-level disconnection between UI-related offscreen computational effort and visual rendering, inherent to dual-thread rendering architectures employed across modern mobile platforms such as Android, iOS, and OpenHarmony. While this architectural pattern is shared across platforms, our current implementation and evaluation focus on OpenHarmony. However, this is hard to detect automatically due to a lack of fine-grained performance measurements and detection methods. To address the challenges, we propose *HapPRDetection* that contains a fine-grained performance profiler that can sample *CPU Retired Instructions*, the CPU instructions that have completed their execution and are no longer in the pipeline, and algorithms for automated detection of *Phantom Rendering* through differential analysis and hierarchical attribution. Our approach advances performance analysis methodology by bridging the semantic gap between fine-grained computational measurements (*i.e.*, the number of *CPU Retired Instructions* at the function level) and high-level rendering behavior. Through our evaluation of the top-22 real-world mobile applications by download volume in OpenHarmony with 193 test steps, we show that *Phantom Rendering* issues occur in 19 test cases across 8 applications and that they range up to 40% of wasted CPU instructions in each problematic operation. We provide new insights into mobile rendering efficiency and our detection strategy offers practical solutions to identify and resolve *Phantom Rendering* during the mobile development loop. Our approach and implementation are available at https://github.com/SMAT-Lab/PhantomRendering.git.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Phantom Rendering, Mobile Performance Analysis, OpenHarmony

*Both authors contributed equally to this research.
†Corresponding author.

Authors' Contact Information: Zhihao Lin, Beihang University, Beijing, China, mathieulin@buaa.edu.cn; Mingyi Zhou, Beihang University, Beijing, China, zhoumingyi@buaa.edu.cn; Bo Sun, Huawei, Wuhan, China, bb.sunbo@huawei.com; Han Hu, Huawei Hong Kong Research Center, Hong Kong, Hong Kong, agmaiofhuhan@gmail.com; Gang Fan, Huawei Hong Kong Research Center, Hong Kong, Hong Kong, fan.gang.cn@gmail.com; Li Li, Beihang University, Beijing, China, lilicoding@ieee.org.



ACM 2994-970X/2026/7-ARTFSE073
https://doi.org/10.1145/3797101

ACM Reference Format:
Zhihao Lin, Mingyi Zhou, Bo Sun, Han Hu, Gang Fan, and Li Li. 2026. Phantom Rendering Detection: Identifying and Analyzing Unnecessary UI Computations. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE073 (July 2026), 22 pages. https://doi.org/10.1145/3797101

1 Introduction

Modern mobile platforms, including Android, iOS, and emerging systems such as OpenHarmony, exhibit the same common phenomenon: applications that appear visually smooth to users, yet consume excessive energy and generate significant heat [6, 8, 20, 32, 36]. Traditional UI performance metrics such as UI jank and frame drops focus on high-level smoothness (*e.g.*, high latency), but it is difficult to capture fine-grained computational inefficiencies [28]. For example, the performance issue in UI rendering has complex causes, usually originating in inefficient code functions.

Through analyzing the UI rendering mechanism on mobile platforms, applications sometimes perform unnecessary offscreen computations on the UI thread but do not produce the corresponding visual rendering [25, 29]. We define this "unnecessary computation" as *Phantom Rendering*, a type of inefficient computation in mobile applications. Importantly, Phantom Rendering is distinct from the previously studied redundant rendering in both phase and granularity. Redundant rendering typically addresses inefficiencies at the GPU or Render Thread stage, such as overdrawing unchanged content that still produces visual output. In contrast, Phantom Rendering occurs upstream at the UI Thread stage, where substantial layout and measure computations are performed but discarded before ever reaching the render thread. Our primary contribution lies not in identifying a new problem class, but in proposing a systematic, hardware-counter-based detection methodology that can automatically quantify and localize this upstream computational waste at the function level. This phenomenon is caused by the dual-thread rendering architecture employed across modern mobile platforms, where UI thread computation can become decoupled from actual rendering output. Although mobile operating systems legitimately skip rendering when content remains unchanged or surfaces are occluded, the high rate of *Phantom Rendering* will cause serious performance issues that affect all mobile platforms. Excessive *Phantom Rendering* consumes significant computational resources that will affect the latency and battery life.

However, this problem is usually caused by bad practice at the function level in the source code. Identifying the problem in the source code requires a fine-grained low-level performance profiler. Existing approaches and tools fail to solve this problem for several reasons. First, the ❶ *attribution problem*: existing approaches can detect computational overhead but are unable to distinguish productive computation from wasteful *Phantom Rendering*, since both exhibit CPU activity in performance traces. Second, the ❷ *measurement problem*: traditional metrics focus on user-perceivable performance rather than computational efficiency, leaving energy waste without visual symptoms in a blind spot. Third, the ❸ *causality problem*: the temporal and architectural separation between UI computation and rendering output in dual-thread systems makes it difficult to establish causal relationships between computational work and visual results. These limitations show that *Phantom Rendering* detection requires a new performance analysis framework that can bridge the semantic gap between fine-grained computational measurements and high-level rendering behavior.

To address these challenges, we propose *HapPRDetection*, a framework for detecting and diagnosing *Phantom Rendering*. It integrates a fine-grained performance profiler that can sample *CPU Retired Instructions* and algorithms to detect *Phantom Rendering* using differential analysis and hierarchical attribution. *CPU Retired Instructions* measures the number of CPU instructions that have completed execution and retired from the processor pipeline. By precisely attributing *CPU*

Retired Instructions to specific functions and correlating them with rendering outcomes, *HapPRDetection* can quantify and localize computational waste at the function level. As shown in Table 1, *HapPRDetection* correlates fine-grained performance data with actionable optimization guidance, allowing developers to identify and resolve *Phantom Rendering* issues without requiring specialized performance engineering knowledge.

Table 1. Comparison of mobile performance analysis tools

Tool	Auto	Energy	Granularity	Root-cause	CI/CD
HapPRDetection	✓	✓ (proxy)	Function/Module	✓	✓
Android Studio Profiler / Perfetto [38]	×	×	Thread/Frame	×	×
Instruments (iOS) [1]	×	✓ (log)	Function/Method	×	×
APM SDK (AndroidGodEye [24])	✓	×	App/Page	×	✓
PerformanceSuite (Booking.com) [39]	✓	×	Screen/Feature	×	✓
Static analyzers (Lint, Infer [7, 12])	✓	×	Line/Pattern	×	✓
Android energy tools (Power Rails [13])	×	✓ (direct)	Device/Subsys	×	×
External power meter (Monsoon) [21]	×	✓ (direct)	Device	×	×

Note: Tools such as Perfetto and Instruments support manual root-cause analysis through expert trace inspection. The "Root-cause" column indicates automated root-cause localization capability.

HapPRDetection creates an automated analysis pipeline that connects the fine-grained hardware-level *CPU Retired Instructions* with high-level UI behavior analysis on mobile platforms. *HapPRDetection* precisely measures *CPU Retired Instructions*, and correlates these data with the rendering states of the UI. The framework uses the common dual-thread rendering architecture found in modern mobile platforms, making the detection methodology conceptually applicable to platforms sharing this dual-thread architecture, although our current implementation and evaluation are conducted on OpenHarmony. By attributing instruction costs to specific functions, our framework achieves automated detection, classification, and root cause localization of *Phantom Rendering* at the function level. This reduces the manual effort required for Phantom Rendering diagnosis by automating the correlation between frame-level events and function-level computational costs. While an expert engineer could manually perform similar analysis using tools such as Perfetto—by visually scanning timelines for doFrame slices, verifying the absence of render thread correlation, identifying CPU time slices, and manually aggregating sampling stacks—HapPRDetection automates this multi-step workflow into a single analysis report, providing direct function-level recommendations integrated into development workflows.

Our evaluation demonstrates that *HapPRDetection* reduces diagnostic time from expert hours to automated minutes while effectively identifying *Phantom Rendering* issues with substantial performance impact. Through the analysis of the top-22 real-world mobile applications by download volume with 193 test steps, we found that *Phantom Rendering* issues occur in 19 test cases across 8 applications, with severe cases causing up to 40% computational waste. The framework integrates into CI/CD pipelines across different mobile development environments, changing performance optimization from reactive post-development fixes to proactive continuous prevention during development. The primary contributions of this paper are as follows:

(1) We characterize the performance issue of *Phantom Rendering* as a frequent inefficient operation in mobile applications, summarizing the pattern of this performance issue in dual-thread rendering architectures and demonstrating its prevalence and impact through quantitative analysis.
(2) We propose a fine-grained performance profiler that can sample the *CPU Retired Instructions*, which provides a low-level indicator for code efficiency at the function level.

(3) We propose *HapPRDetection*, a framework for the automated detection of *Phantom Rendering* through differential analysis and hierarchical attribution, and validate it through comprehensive evaluation in top-22 real-world applications, demonstrating the effectiveness and utility of our framework. Our approach is compatible with the existing testing workflows.

The remainder of this paper is organized as follows. Section 2 introduces the background and related work, provides context for the detection of *Phantom Rendering* in mobile platforms, and compares existing performance analysis tools. Section 3 presents our motivation study, which demonstrates the prevalence and impact of *Phantom Rendering* through controlled experiments. Section 4 details the architecture and methodology of *HapPRDetection*, including our four-module framework and the core algorithms. Section 5 presents our evaluation of real-world applications, validating the effectiveness and practical utility of the framework. Finally, Section 6 discusses threats to validity and limitations, and Section 7 concludes the article with future research directions.

2 Background

2.1 Mobile Platform Rendering Architecture

Contemporary mobile platforms universally adopt a dual-thread rendering architecture that separates UI computation from actual rendering execution. This architectural pattern, implemented on Android, iOS, OpenHarmony, and other mobile operating systems, consists of two key components:

(1) **UI Thread**: Handles user interactions, executes business logic, computes layout parameters, and generates render trees within the application process.
(2) **Render Thread**: Runs as an independent system service or thread, receiving render trees from UI threads and performing actual GPU-accelerated rendering operations.

This dual-thread architecture, while enabling efficient resource utilization across mobile platforms, introduces various performance challenges that manifest consistently across different mobile operating systems. Traditional graphics research has studied overdrawing, where pixels are drawn multiple times within a single frame due to overlapping geometry or suboptimal rendering order [17, 43]. Similarly, redundant rendering refers to the unnecessary redrawing of unchanged content [26]. These rendering inefficiencies occur during the actual rendering process and can be detected through GPU performance counters and pixel fill rate analysis available on modern mobile hardware. Furthermore, mobile platforms commonly experience performance issues including excessive frame latency [45], where the time between user input and visual feedback becomes noticeable, and frame drops [33], where the system skips rendering frames to maintain synchronization with display refresh rates. These issues have been extensively studied in the graphics and mobile performance literature, with various platform-specific and cross-platform optimization techniques proposed to address them.

2.2 Mobile Performance Analysis and Energy Measurement

Traditional performance engineering in mobile applications [2, 35, 37, 42] has established many useful indicators, such as FPS, frame-time percentiles, input latency, start time, and jank rate. These indicators guide manual diagnosis and adjustment, as they can describe the smoothness and responsiveness of apps; therefore, they promote best practices in profiling and regression testing. However, manual diagnosis and tuning are expensive, especially with large codebases. When the goal is automated diagnosis and optimization on scale, relying on such subjective or context-dependent indicators becomes problematic. Their interpretation often depends on workload characteristics, user interaction patterns, device heterogeneity, and app-specific thresholds, which limits reproducibility and hinders automatic root-cause attribution.

Energy consumption analysis has emerged as a critical aspect of optimization of mobile performance [11, 41]. Direct power measurement approaches using external hardware [21] provide accurate energy readings but lack fine-grained attribution to specific application components. Software-based energy estimation techniques [13] offer better integration with development workflows but often require complex modeling and calibration. Recent research has explored the use of hardware performance counters as energy proxies [4, 5, 19, 22], leveraging the strong correlation between computational activity and power consumption to enable precise energy attribution.

2.3 Mobile Performance Analysis Tools

The mobile development ecosystem provides various performance analysis tools with different capabilities and limitations. Platform-specific tools such as Android Perfetto [38] and iOS Instruments [1] offer comprehensive system tracking and detailed performance insights, but require expert knowledge for effective interpretation and root cause analysis. These tools excel at capturing system-wide performance data, but often lack the semantic understanding needed to distinguish productive computation from wasteful overhead. Application Performance Monitoring (APM) solutions [24, 34, 39] provide automated monitoring capabilities suitable for production environments, but typically focus on high-level metrics and user-perceivable performance issues rather than computational efficiency. Static analysis tools [7, 10, 12] can identify potential performance problems at the code level but cannot detect run-time inefficiencies that arise from complex interaction patterns and system dynamics.

3 Motivation

We have established *Phantom Rendering* as an important performance issue in Section 1, and real-world evidence demonstrates its practical importance. Mobile device users frequently report unexpected battery drain and overheating, as shown in Figure 1. Even if the user does not interact with them, such apps can consume excessive power and generate significant heat, affecting battery life, device performance, and user satisfaction. These user complaints indicate that background computations are prevalent and impactful, even without user interaction. This real-world evidence supports our hypothesis that *Phantom Rendering* is widespread in production applications and has tangible user-facing consequences.

To quantify the impact of *Phantom Rendering* and understand detection challenges, we constructed controlled experiments in OpenHarmony with four canonical interaction scenarios: ❶ image browsing with scroll operations, ❷ like interaction with state changes, ❸ pinch zoom gestures on images, and ❹ comment scrolling with dynamic loading. Each scenario was implemented in two versions: a normal implementation following standard practices and an abnormal implementation that intentionally triggers *Phantom Rendering* by performing substantial UI computations that are immediately canceled or overridden.

We monitored these scenarios using OpenHarmony's built-in profiling tools, tracking device-level metrics such as device temperature, CPU usage, and memory usage. The profiler output data verifies the large difference in resource consumption between normal and *Phantom Rendering* scenarios, confirming the computational overhead observed in our problem definition. However, our investigation identified an important detection challenge: current performance analysis frameworks have the ability to detect symptoms of *Phantom Rendering* (high CPU usage, increased temperature), but cannot identify or quantify particular forms of computational waste patterns. Table 2 displays the system-level measurements.

The experimental results demonstrate the severity of the impact of *Phantom Rendering*. Process CPU utilization increases by 10.51-28.3 percentage points across all scenarios, with comment scrolling (+28.3%) and image browsing (+25.4%) showing the most dramatic increases. The device

Fig. 1. Real-world user complaints on social media about mobile device overheating and excessive battery drain during app usage, despite minimal user interactions.

Table 2. System profiler measurements across interaction scenarios using OpenHarmony built-in profiling tools, comparing the normal mode and the *Phantom Rendering* mode

Scenario	**Process CPU (N → P)**	**Temperature (N → P)**	**Memory Usage (N → P)**
Image browsing	0.67% → 26.07%	35.4°C → 36.0°C	118.1MB → 132.2MB
Like interaction	9.27% → 24.15%	35.9°C → 36.5°C	139.6MB → 137.3MB
Pinch-zoom operation	10.53% → 21.04%	36.6°C → 36.9°C	142.3MB → 146.1MB
Comment scrolling	21.06% → 49.36%	37.9°C → 40.0°C	168.2MB → 246.5MB

temperature increases consistently by 0.3-2.1 ° C, with the comment scrolling showing the highest increase in temperature (+2.1 ° C). Memory usage shows substantial changes, particularly in comment scrolling, where it increases by 78.3MB. These metrics directly correlate with user-reported symptoms of device overheating and excessive battery consumption shown in Figure 1. Although our measurements confirm that *Phantom Rendering* causes substantial computational overhead, they expose a critical limitation in current performance analysis approaches. In our controlled experiment, we can identify the impact of *Phantom Rendering* by comparing normal versus abnormal implementations: the dramatic difference from 0.67% to 26.07% CPU usage clearly indicates waste. However, this comparison-based detection is impossible in production environments. A developer examining a real application cannot determine whether 15% CPU usage represents efficient computation or contains significant *Phantom Rendering* waste, as there is no baseline for comparison. This fundamental limitation prevents developers from identifying *Phantom Rendering* patterns that can be hidden within seemingly normal performance profiles, creating an urgent need for detection frameworks that work without baseline comparisons.

This motivation study establishes three key findings that drive our framework's design: ❶ *Phantom Rendering* causes measurable computational overhead (10.51-28.3 percentage points of CPU increase), which is directly correlated with user-reported battery and heating issues, ❷ current performance analysis tools can detect symptoms but cannot identify root causes or quantify waste, and ❸ deployed environments lack baseline comparisons, making traditional performance analysis insufficient for detection of *Phantom Rendering*. These results indicate the urgent need for specialized automated frameworks that can detect and localize *Phantom Rendering* patterns without requiring expert interpretation or baseline comparisons.

4 Approach

4.1 Overview

Our framework addresses the challenge of detecting and quantifying *Phantom Rendering* by combining multi-dimensional data collection with differential analysis. The core insight is that *Phantom Rendering* detection requires correlating computational work with visual output across temporal and architectural separation in dual-thread rendering systems. As illustrated in Figure 2, *HapPRDetection* consists of four interconnected modules: Scripts Generation Module (SGM), Random Click Module (RCM), Performance Exploration Module (PEM), and Issue Detection Module (IDM). *HapPRDetection* combines hardware-level precision through *CPU Retired Instructions* with frame-level behavioral analysis to establish causal relationships between computational effort and visual output.

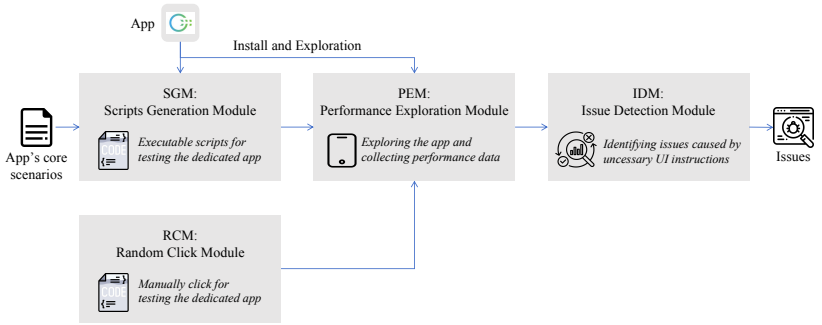


Fig. 2. The overview design of our *Phantom Rendering* detection framework

HapPRDetection addresses the three challenges identified in Section 1. The *attribution problem* requires distinguishing productive from wasteful CPU usage. Productive computation generates user-visible results. *Phantom Rendering* wastes CPU cycles without rendering the output. However, standard profilers cannot distinguish between them. We solve this through differential analysis that compares computational patterns between phantom and normal rendering scenarios. The *measurement problem* requires objective quantification of computational efficiency beyond user-perceivable metrics, and we address this by establishing *CPU Retired Instructions*, which provides hardware-level precision with fine-grained function-level attribution [4, 19, 40]. The *causality problem* involves establishing relationships between UI computation and rendering output across

temporal and architectural separation; we resolve this through synchronized multi-dimensional data collection that captures both computational events and rendering states, enabling correlation analysis [5, 14, 23].

Although isolated instances of *Phantom Rendering* can be normal across all mobile platforms (e.g., when UI legitimately skips drawing unchanged content), a sustained high rate of *Phantom Rendering* indicates abnormal UI computational waste regardless of the underlying mobile operating system. Our framework specifically targets these high rate of *Phantom Rendering* patterns that represent significant energy waste in mobile applications. We take advantage of common characteristics of mobile platforms: *CPU Retired Instructions* analysis to quantify computational effort and energy consumption at the hardware level using standard performance counters, and frame-level analysis to detect scenarios where the computation of the UI thread occurs without the corresponding rendering output, a pattern that is observable across mobile platforms due to their shared dual-thread rendering architecture. This approach enables the precise attribution of computational waste to specific UI operations while measuring their energy impact through hardware-agnostic *CPU Retired Instructions* analysis.

4.2 Framework Architecture

To address the challenges of *Phantom Rendering* detection, we propose *HapPRDetection*. We integrate a Script Generation Module (SGM) and the Random Click Module (RCM), ensuring complete coverage of *Phantom Rendering* triggers through controlled and naturalistic interaction patterns. The Performance Exploration Module (PEM) implements the measurement by synchronizing hardware-level instruction counting with frame-level behavioral analysis. The Issue Detection Module (IDM) applies our main algorithms to input apps and performs root cause localization for the found *Phantom Rendering* through hierarchical analysis. This integrated framework enables systematic detection of *Phantom Rendering* while maintaining a resolution level suitable to guide user actions for optimization at the function level.

4.2.1 Scripts Generation Module and Random Click Module. The Scripts Generation Module (SGM) addresses the challenge of reproducible *Phantom Rendering* detection by generating structured test sequences that systematically exercise different UI interaction patterns. This module ensures full coverage of known *Phantom Rendering* triggers while maintaining experimental control for precise measurement. SGM employs template-driven test generation that can be adapted to different mobile platforms and applications, providing the systematic foundation needed for reliable *Phantom Rendering* analysis.

The Random Click Module (RCM) addresses the limitation that *Phantom Rendering* often emerges from unpredictable user interaction patterns that cannot be captured through predefined scenarios. RCM employs probabilistic interaction modeling to generate user behavior patterns, including overlapping interactions, varied timing, and complex gesture sequences [3, 27, 31]. The combination of the SGM approach and RCM's probabilistic simulation ensures coverage of both reproducible and emergent *Phantom Rendering* scenarios, addressing the challenge that *Phantom Rendering* occurrence is correlated with interaction complexity and unpredictability.

Both of these modules serve to test the detection of *Phantom Rendering* but in different ways. The Scripts Generation Module (SGM) allows users to reproduce and control *Phantom Rendering* in order to help debug individual problems related to it and verify whether the cause has been fixed. The Random Click Module (RCM), on the other hand, provides broader coverage without requiring manual script creation, enabling faster testing across different usage scenarios and helping to discover unexpected *Phantom Rendering* triggers that systematic testing might miss.

4.2.2 Performance Exploration Module. The Performance Exploration Module (PEM) implements our measurement theory through synchronized multi-dimensional data collection that addresses the causality problem in *Phantom Rendering* detection. PEM employs three specialized analyzers that collectively capture the computational and behavioral aspects of mobile rendering: performance analysis captures *CPU Retired Instructions* data with function-level attribution using hardware performance counters, frame analysis identifies *Phantom Rendering* episodes by correlating UI thread computation with render thread output, and synchronization analysis detects display coordination issues that contribute to computational waste. The module uses multi-threaded coordination that enables a precise temporal correlation between computational events and rendering outcomes, as illustrated in Figure 3. Four synchronized data streams capture complementary aspects of the rendering process: hardware-level instruction retirement monitoring provides microsecond-resolution computational measurements, UI interaction tracking captures user-initiated events and state transitions, hierarchical UI state analysis enables before/after comparison of rendering contexts, and render thread monitoring tracks actual visual output production. This coordinated approach solves the causality problem by establishing precise temporal relationships between computational work and visual results across the dual-thread architecture.

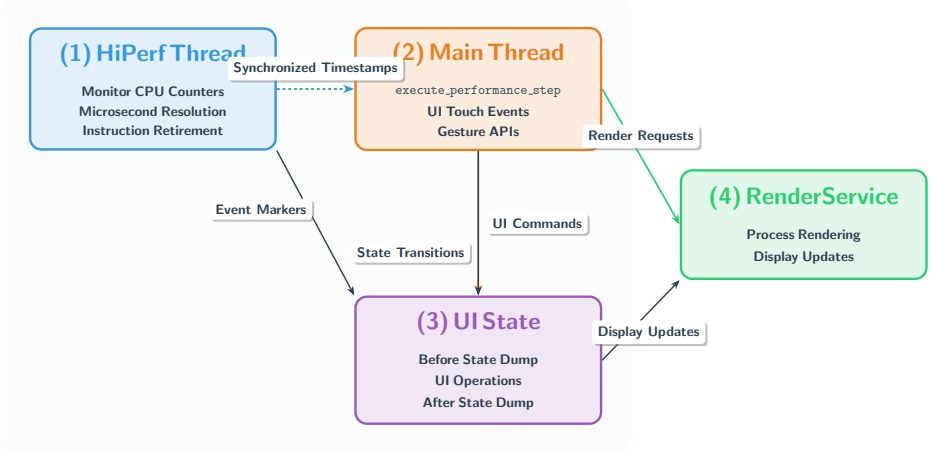


Fig. 3. Multi-threaded coordination architecture for performance data collection

PEM solves the problem of attribution by correlating *CPU Retired Instructions* measurements with thread-level activity patterns, enabling precise cost attribution to specific code modules and functions. The approach is able to reconstruct detailed execution contexts by correlating hardware-level instruction retirement events with software-level call stacks. This provides the granular attribution needed for actionable optimization guidance. Symbol resolution transforms machine-level performance data into developer-accessible function-level information [9, 44], where the trace can be directly linked from *Phantom Rendering*. This allows identification of the corresponding code location responsible for wasted computation.

4.2.3 Issue Detection Module. The Issue Detection Module (IDM) implements our core algorithmic contributions through differential analysis and hierarchical attribution methodologies. IDM addresses the attribution problem by distinguishing at the instruction level whether computing is legitimate UI computation from wasteful computing due to *Phantom Rendering*. It provides the

ability to detect the frames where a computational budget is spent computing but no corresponding visual output is produced.

The differential analysis algorithm operates through frame-level coordination analysis that examines the temporal relationship between the computational activity of the UI thread and the production of the thread output, as shown in Figure 4. We classify run-time behavior into four categories: ❶ normal frames with coordinated computation and rendering, ❷ delayed frames indicating performance bottlenecks, ❸ *Phantom Rendering* frames with computation but no visual output, and ❹ synchronization anomalies with timing deviations. The algorithm performs detection of cases where substantial UI thread computation occurs without the corresponding visual output, directly capturing computational waste patterns. Specifically, this detection relies on objectively measurable binary states: either the render service produces visual output for a given UI computation, or it does not. This binary nature eliminates subjective interpretation in distinguishing productive computations from *Phantom Rendering* waste. It is important to note

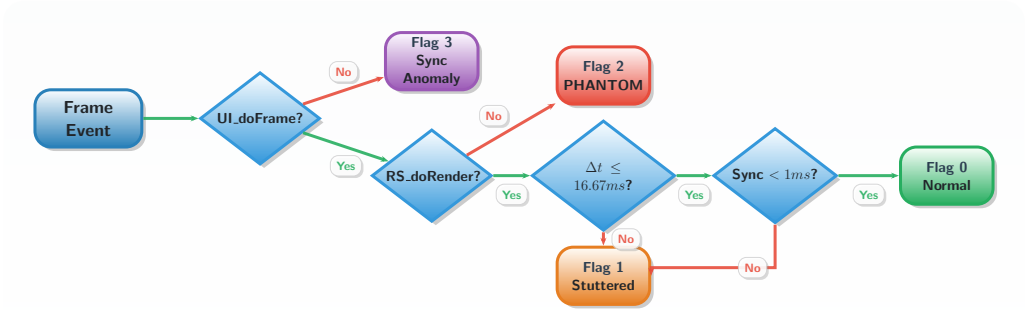


Fig. 4. Frame classification decision tree for *Phantom Rendering* detection

that not all phantom frames indicate application defects. Mobile operating systems legitimately skip rendering when content remains unchanged or surfaces are occluded. In such cases, the UI thread typically performs only a lightweight check with negligible instruction counts, which our framework filters out as benign behavior. Our detection specifically targets frames where the UI thread executes substantial computation (reflected by high *CPU Retired Instructions* counts) but produces no visual output. For instance, in one detected case, an animation component continued executing complex layout logic even after the user had navigated to a different page. Although the Render Service correctly culled the drawing, which is a valid OS optimization, the UI thread's continued heavy computation constitutes measurable energy waste. This cost-based distinction ensures that our detection targets genuine computational inefficiency rather than normal OS-level frame management. Our differential analysis methodology quantifies the impact of *Phantom Rendering* by comparing computational costs between phantom and normal rendering scenarios. The approach employs Equation 1 to calculate differential weights that identify functions that exhibit excessive computational overhead during *Phantom Rendering* episodes.

$$W_{\text{sym}} = \left(\frac{I_{\text{phantom}} - I_{\text{normal}}}{D_{\text{stack}}} \right) \cdot W_{\text{comp}}, \tag{1}$$

where $I_{phantom}$ and I_{normal} represent CPU instruction retirement counts in phantom and normal modes, respectively. This differential weight calculation enables the systematic identification of

computational waste sources by isolating the additional computational overhead specific to the scenarios of *Phantom Rendering*.

Our hierarchical attribution technique tackles the challenge of tracing *Phantom Rendering* sources across various abstraction levels, as illustrated in Figure 5. Our framework first attributes computational waste from high-level processes down to the symbol level of a particular function from the top to bottom by following four layers: process-level attribution identifies applications contributing to *Phantom Rendering* overhead, thread-level attribution isolates specific threads responsible for excessive computation, file-level attribution pinpoints source files containing problematic functions, and symbol-level attribution identifies exact functions responsible for computational waste. This multi-layered framework ensures full coverage while also maintaining accurate quantification of consumed computational at each abstraction level.

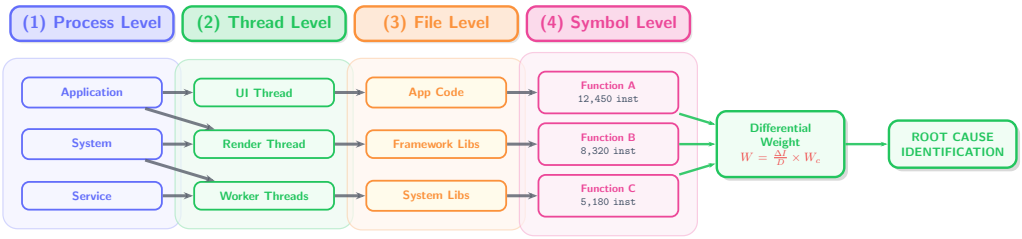


Fig. 5. Four-layer hierarchical attribution mechanism for *Phantom Rendering* root cause analysis

As shown in Figure 6, the final result of this process consists of reports "function payload". This report provides developers with code-level feedback on *Phantom Rendering* optimization, identifying resource-consuming functions through differential comparisons, and quantifying their computational cost in *CPU Retired Instructions*. Through this report, *HapPRDetection* provides explicit links between *Phantom Rendering* issues and specific code locations responsible for energy inefficiency.

Function	Instructions	Thread	Process	Affiliation File	Classification
performRealImageWorkBuf	1.30×10^9	Main	com.example	ImageBrowsePage.ts	APP_ABC
anonymous	7.38×10^8	Main	com.example	ImageBrowsePage.ts	APP_ABC
performRealSwipeWorkBuf	4.49×10^8	Main	com.example	ImageBrowsePage.ts	APP_ABC
rerender	4.27×10^8	Main	com.example	ImageBrowsePage.ts	APP_ABC
OHOS::Ace::Framework::JsRuntimeCall	4.05×10^8	Main	/system/lib64	libace_compatible.z.so	OS_Runtime
currentIndex	3.54×10^8	Main	com.example	ImageBrowsePage.ts	APP_ABC
imageData	3.41×10^8	Main	com.example	ImageBrowsePage.ts	APP_ABC
AddHitraceMeterMarker	2.61×10^8	Main	/system/lib64	libtrace_meter.so	SYS_SDK

Fig. 6. Function payload analysis showing computational cost attribution for *Phantom Rendering*

4.3 Implementation Details

Our OpenHarmony implementation integrates with Hypium testing framework [16] and HiPerf profiling tools [15]. The generation of tests employs a base class with `execute_performance_step()` API for coordinated UI interaction and performance monitoring. Random interaction uses probabilistic models with OpenHarmony UI automation APIs. Performance monitoring implements

three analyzers: *PerfAnalyzer* for CPU flame graph analysis using HiPerf counters, *EmptyFrameAnalyzer* for phantom frame statistics, and *VSyncAnomalyAnalyzer* for display synchronization issues. Multi-threaded coordination uses four parallel streams: HiPerf data collection at microsecond resolution, main thread UI interaction, UI tree state capture via accessibility APIs, and render thread monitoring through RenderService traces. Data processing converts raw HiPerf/HiProfiler traces to SQLite through *trace_streamer*, with symbol resolution using WebAssembly-accelerated C++ demangling [9, 18]. Frame classification uses a decision tree with flags based on `UI_doFrame`/`RS_doRender` events: Normal (both present, $\Delta t \leq 16.67$ms), Stuttered (both present, $\Delta t > 16.67$ms), Phantom (UI only), and Sync Anomaly (timing deviation $\geq$ 1ms).

$$\text{Flag} = \begin{cases} 0 \text{ (Normal)}, & \exists \texttt{UI\_doFrame} \wedge \exists \texttt{RS\_doRender} \wedge \Delta t \leq 16.67\text{ms}, \\ 1 \text{ (Stuttered)}, & \exists \texttt{UI\_doFrame} \wedge \exists \texttt{RS\_doRender} \wedge \Delta t > 16.67\text{ms}, \\ 2 \text{ (Phantom)}, & \exists \texttt{UI\_doFrame} \wedge \nexists \texttt{RS\_doRender}, \\ 3 \text{ (Sync Anomaly)}, & \text{timing deviation} \geq 1\text{ms}. \end{cases} \tag{2}$$

Importantly, *Phantom Rendering* detection is objectively deterministic: a frame is classified as phantom if and only if `UI_doFrame` events occur without corresponding `RS_doRender` events, representing a clear binary state where UI computation happens but no visual output is produced. This binary classification eliminates ambiguity in distinguishing productive computation from wasteful *Phantom Rendering*. The threshold-based analysis applies to determine when the frequency of phantom frames indicates abnormal behavior, not to the classification of individual frames themselves. We configure a *Phantom Rendering* ratio threshold of 3% for anomaly classification, informed by both theoretical modeling and expert validation. We formalize a Phase Drift model to establish the expected noise floor of phantom frames in a well-functioning dual-thread system. The baseline noise rate θ is modeled as:

$$\theta = \frac{f_{\text{drift}} + f_{\text{noise}}}{F_{\text{vsync}}} \tag{3}$$

where $F_{\text{vsync}} = 60$ Hz is the display refresh rate, $f_{\text{drift}} \approx 1$ Hz represents the maximum benign beat frequency due to phase misalignment between the UI and render threads, and $f_{\text{noise}} \approx 0.8$ Hz accounts for stochastic OS scheduling jitter. This yields $\theta \approx 3\%$, below which phantom frames are statistically attributable to system-level noise rather than application defects. This theoretical result was further validated through consultation with six senior performance engineers (each with 10+ years of experience), who confirmed that phantom rates below 3% are typically non-actionable system jitter. At a refresh rate of 60 fps, the 3% threshold translates to approximately 2 wasted computation frames per second. Furthermore, our most critical findings (10%–40% phantom rates) are well above this threshold and remain robust to minor threshold variations. Although individual *Phantom Rendering* frames may represent normal system optimization behavior, a sustained rate exceeding this noise floor constitutes a persistent energy consumption issue that warrants developer attention, particularly in long-running applications such as video streaming or social media feeds. Finally, VSync anomaly detection employs frequency analysis with severity thresholds of 0.10 and 0.30.

5 Evaluation

In this section, we investigate three research questions to understand *Phantom Rendering* in real-world mobile applications. We investigated the prevalence and patterns of *Phantom Rendering* in different categories of applications and user interaction scenarios, quantify its performance and energy impact on mobile devices, and validate the effectiveness of *HapPRDetection*. Our evaluation includes the top-22 real-world applications by download volume in the OpenHarmony application

store that cover 12 categories with 193 test steps, providing robust evidence of the widespread nature of *Phantom Rendering* and demonstrating the practical value of systematic detection and quantification.

5.1 Experimental Setup

5.1.1 Testing Environment. We conducted the measurement in an OpenHarmony stable version(5.0.1.120.SP3) mobile phones. We collect performance data using the built-in HiPerf and HiProfiler tools, with consistent configuration parameters in all test runs to ensure reproducibility. In addition, we designed a controlled testing environment that included standardized testing procedures, consistent data collection methods, and unified measurement tool calibration procedures. We collected 193 test step records across 22 different applications covering 12 distinct categories, providing extensive coverage of real-world usage scenarios, from lightweight reading applications to complex multimedia platforms. We keep the system conditions for data collection consistent, such as the device state and controlled background services.

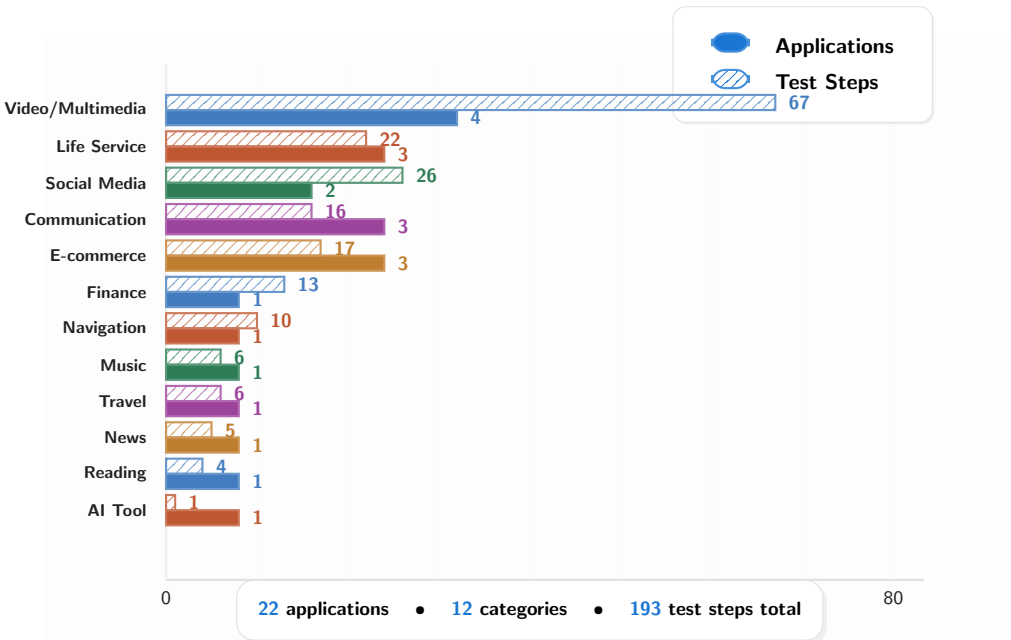


Fig. 7. Evaluation Applications Summary by Category

5.1.2 Testing Methodology. We used two complementary testing approaches to comprehensively evaluate *Phantom Rendering* issues in different usage scenarios. ❶ Batch Testing (Automated Script-Based): We developed automated test scripts for several main applications, including Financial App A, Social Media App A, Social Commerce App A, E-commerce App A, Video Platform App A, and E-commerce App B. This approach ensures reproducible and consistent testing conditions with standardized interaction sequences. Batch testing covers 95 test step records across diverse scenarios, from financial transactions to video streaming, providing controlled baseline measurements for *Phantom Rendering* detection and enabling systematic performance comparison. ❷Manual Testing

(Realistic User Interaction): To capture authentic user behavior patterns, we conducted manual testing sessions in various applications, including social media (Social Media App B), communication (Messaging App A), navigation (Navigation App A), food delivery (Food Delivery App A), and office collaboration (Enterprise Collaboration App A). The testers performed realistic interactions including browsing, searching, messaging, and location-based operations. This approach covers 98 test step records and reveals *Phantom Rendering* issues that occur under real-world usage conditions, providing insight into performance challenges that automated testing might miss. The combination of these two testing methodologies provides comprehensive coverage that spans from highly controlled environments to realistic usage scenarios.

5.2 Research Questions

To validate our findings about *Phantom Rendering* and demonstrate the effectiveness of *HapPRDetection*, we investigate the following research questions:

- **RQ1:** How common is *Phantom Rendering* in real-world mobile applications, and what types of operations trigger it most frequently?
- **RQ2:** What impact does *Phantom Rendering* have on the performance of the mobile application and the consumption of energy?
- **RQ3:** How effective is our detection method in identifying *Phantom Rendering* and localizing root causes?

These questions systematically evaluate our contributions. RQ1 characterizes the phenomenon of *Phantom Rendering* by examining its prevalence and patterns in 22 real-world applications. RQ2 quantifies the actual performance and the energy impact of *Phantom Rendering* to demonstrate its significance. RQ3 evaluates the effectiveness of *HapPRDetection* through validation of precision of accuracy, consistency, and attribution.

5.3 Results

5.3.1 Phantom Rendering Prevalence and Patterns (RQ1). To characterize the phenomenon of *Phantom Rendering* in real-world scenarios, we systematically analyzed its occurrence patterns in 22 different OpenHarmony applications in 12 categories, examining 193 test steps to understand when and why *Phantom Rendering* occurs.

Our analysis reveals that *Phantom Rendering* is a widespread phenomenon that occurs 19 times out of 193 steps in various mobile applications. As shown in Figure 8, the severity distribution shows significant variation: 174 low-impact cases (<3% phantom rate), 7 medium impact (3- 5%), 4 high impact (5- 10%) and 8 critical cases (>10%). The most severe cases show phantom rates up to 40. 15%, indicating that almost half of computational work does not produce visual output in certain interaction scenarios.

The occurrence of *Phantom Rendering* is mainly driven by specific interaction patterns. Content scrolling operations show the highest *Phantom Rendering* rates, where continuous data streaming combined with rapid viewport movement creates scenarios where UI threads perform substantial layout computation for content that may be quickly discarded or invalidated by up-to-date incoming data. Dynamic content updates are another common source of *Phantom Rendering*, including real-time notifications, live data feeds, and location-based information updates. These activities will inevitably trigger frequent UI state changes, and usually result in rendering work being performed but subsequently discarded due to rapid data changes. Operations such as interactive animations and transitions also contribute to *Phantom Rendering*. In particular, gesture-based operations such as pinch zoom, swipe navigation, and complex state transitions in the UI, and other operations involving intensive computation for visual effects that may be interrupted or

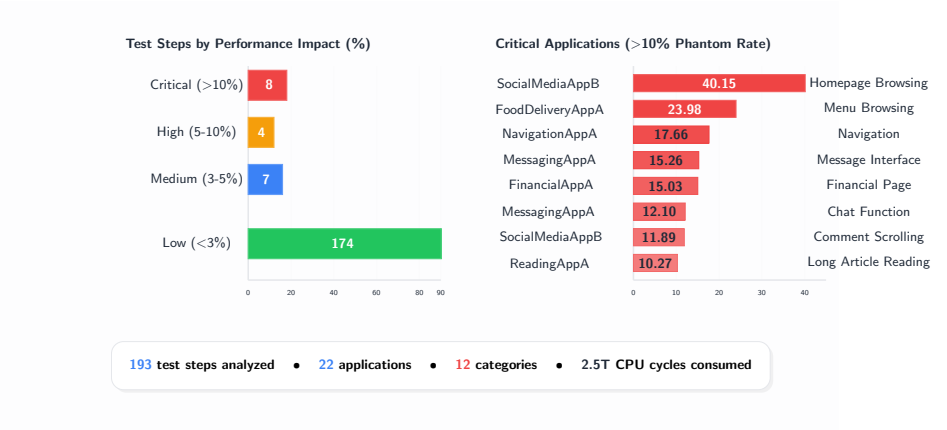


Fig. 8. Detection results overview showing (left) test steps by performance impact levels and (right) critical applications with highest *Phantom Rendering* rates

overridden by subsequent user actions, may lead to wasted rendering cycles. In contrast, static content display and predictable user interactions typically show lower rates of *Phantom Rendering*, as they involve more stable UI states with fewer concurrent updates and less frequent interruptions of the rendering pipeline.

The relation between the complexity of the interaction and the emergence of *Phantom Rendering* is shown by the discrepancies in the experimental procedure. Manual testing reveals detection rates of more frequently *Phantom Rendering* compared to automated batch testing. The reason is that manual testing captures realistic user interaction patterns with higher complexity and unpredictability, while automated testing focuses on standardized repeatable operations. The significant difference validates that *Phantom Rendering* is fundamentally related to the dynamic and unpredictable nature of real-world mobile usage rather than specific application architectures.

Examining specific cases reveals that the most critical *Phantom Rendering* instances span various application categories, confirming that the phenomenon is not limited to particular application types. Social Media App B (40. 15% phantom rate), the restaurant discovery interface of Food Delivery App A (23. 98%), and the navigation display of Navigation App A (17. 66%) represent fundamentally different UI paradigms, yet all exhibit severe rendering inefficiency. This cross-category distribution indicates that *Phantom Rendering* represents a systemic issue inherent in mobile rendering architectures rather than application-specific problems.

> **Answer for RQ1:** Under the studied dual-thread rendering architecture on OpenHarmony, *Phantom Rendering* is a widespread phenomenon in real-world mobile applications, occurring in 19 test cases across 8 applications. Content scrolling operations, dynamic content updates, and interactive animations are the primary triggers, with the most severe cases showing up to 40% *Phantom Rendering* rates. The phenomenon exhibits clear patterns: complex interaction scenarios and frequent state changes significantly increase *Phantom Rendering* occurrence, while static content and predictable interactions show minimal waste.

5.3.2 Performance and Energy Impact Assessment (RQ2). To quantify the actual performance and the energy impact of *Phantom Rendering*, we analyze the computational overhead and resource consumption patterns in all the *Phantom Rendering* episodes detected. This analysis examines

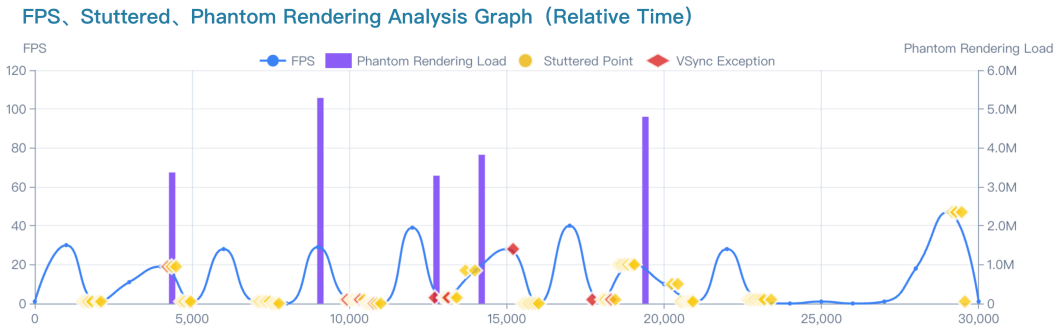


Fig. 9. Performance report of a critical phantom-rendering case showing sustained CPU load with minimal visual output

how *Phantom Rendering* affects system performance, energy efficiency, and overall application responsiveness in real-world usage scenarios.

Table 3. Correlation Between *Phantom Rendering* Rate and Computational Waste

Severity	**Cases**	**Avg *Phantom Rendering* Rate**	**Total Wasted *CPU Retired Instructions***	**Per-Case Wasted**
Critical	8	18.3%	10.0B	1.25B
High	4	6.8%	1.8B	0.45B
Medium	7	4.0%	2.3B	0.33B
Low	174	0.3%	6.2B	0.036B
Total	193	1.36%	20.2B	0.105B

Our quantitative analysis demonstrates a strong positive correlation between *Phantom Rendering* frequency and computational waste. As shown in Table 3, applications with higher *Phantom Rendering* rates consistently exhibit proportionally higher wasted *CPU Retired Instructions*. Critical cases (>10% phantom rate) average 18.3% *Phantom Rendering* rate and waste 1.25B *CPU Retired Instructions* per case, while low-impact cases (<3% phantom rate) average only 0.3% *Phantom Rendering* rate and waste 0.036B *CPU Retired Instructions* per case. The difference in *Phantom Rendering* rate corresponds to a 35× difference in per-case computational waste, establishing a clear quantitative relationship between *Phantom Rendering* frequency and performance impact. The performance impact is further illustrated in Figure 9. *Phantom Rendering* frames sustain high CPU load, showing unnecessary energy waste. The total of 20.2B wasted *CPU Retired Instructions* in all test cases represents measurable energy consumption that directly impacts battery life and thermal management [30]. The general *Phantom Rendering* rate of 1. 36% in all applications masks significant localized inefficiencies that occur in specific usage scenarios. Although most applications operate efficiently under normal conditions, critical cases demonstrate that *Phantom Rendering* can consume substantial computational resources, with some scenarios wasting up to 40% computational work. This validates both the importance of the phenomenon of *Phantom Rendering* and the need for systematic detection and quantification approaches. Critical cases consuming an average of 18.3% of computational resources without visual output. This shows that *Phantom Rendering* significantly degrades the efficiency of the application, particularly in scenarios involving intensive user interactions. These quantified impacts establish concrete optimization targets and justify systematic approaches to *Phantom Rendering* detection and mitigation.

Answer for RQ2: Under the studied conditions on OpenHarmony, there is a strong positive correlation between *Phantom Rendering* frequency and computational waste. Operations with higher *Phantom Rendering* rates consistently exhibit proportionally higher wasted CPU cycles. Critical cases waste 1.25B cycles per case compared to 0.036B for low-impact cases, demonstrating that the frequency of *Phantom Rendering* directly predicts the magnitude of the impact of performance.

5.3.3 Detection Method Effectiveness (RQ3). To evaluate the effectiveness of our differential analysis approach, we assess multiple dimensions of performance: detection consistency, attribution accuracy, and methodological robustness. We conducted controlled experiments using repeated tests of known *Phantom Rendering* cases and validation against constructed *Phantom Rendering* scenarios where the ground truth is established.

Our effectiveness evaluation examines whether our detection method can reliably identify *Phantom Rendering* patterns under different conditions and accurately localize root causes to specific functions. We selected 6 test cases from 4 applications where *Phantom Rendering* had been previously detected, repeating each case 5 times under identical conditions to measure consistency. Furthermore, we validated the precision of the attribution using controlled scenarios with known sources *Phantom Rendering*. Our evaluation demonstrates that the differential analysis approach

Table 4. Reliability Testing Results for Repeated *Phantom Rendering* Detection

App	Cases	Steps	Rounds	Consistency
Social Media App A	2	4	10	100%
E-commerce App A	1	3	5	80%
Social Commerce App A	2	8	10	100%
Financial App A	1	1	5	100%
Overall	**6**	**16**	**30**	**94.4%**

achieves high accuracy and consistency in *Phantom Rendering* detection. As shown in Table 4, across 30 repeated test executions covering 6 cases with 5 repetitions each, our method achieves 94.4% reproducibility across repeated test executions. This metric measures whether the same Phantom Rendering patterns are consistently detected across runs under identical conditions, rather than precision against a ground-truth oracle, which is unavailable for production applications. The non-100% score is attributable to non-determinism in the mobile environment (e.g., slight scheduling variances), confirming the stability of our methodology rather than instability in the phenomenon itself. This high consistency rate validates that our approach identifies genuine, persistent *Phantom Rendering* patterns rather than transient measurement artifacts. The consistency analysis reveals that three applications (Social Media App A, Social Commerce App A, Financial App A) demonstrate perfect 100% detection consistency, indicating that *Phantom Rendering* in these cases stems from deterministic rendering pipeline issues that manifest reliably under consistent testing conditions. The single inconsistent case (E-commerce App A: 80%) resulted from profiling tool limitations during high-frequency UI updates rather than methodological failure, as the phantom frames were still detected but detailed attribution data collection was occasionally incomplete due to system constraints.

Beyond reliability validation, we also demonstrate *HapPRDetection*'s function-level attribution accuracy using constructed test cases from Section 3 and real-world applications. Controlled experiments demonstrate 100% precision in identifying root cause functions across different interaction patterns: `ImageBrowsePage.ts:699` for image browsing, `LikeInteractionPage.ts:665`

for like interactions, `PinchZoomPage.ts:595` for gesture processing, and multiple functions in `CommentScrollPage.ts` for virtual scrolling. These located functions are exactly where we constructed in our demo application to construct the *Phantom Rendering* scenarios. For commercial applications, *HapPRDetection* successfully profiles function-level hotspots during *Phantom Rendering* episodes. Our validation establishes three levels of attribution accuracy: application-level (specific source files and line numbers), framework-level (ArkUI rendering pipeline functions), and system-level (runtime overhead including garbage collection and event handling). Beyond reproducibility, we verified the 19 reported issues through three complementary layers: (1) 100% attribution precision on controlled demo applications with injected Phantom Rendering bugs (Section 3), where the localized functions exactly matched the injected fault locations; (2) correlation of detection traces with screen recordings to confirm the absence of visual output during flagged frames; and (3) industrial confirmation from application developers who acknowledged the reported issues as valid performance defects warranting optimization.

> **Answer for RQ3:** Our detection method demonstrates high effectiveness in multiple evaluation dimensions: 94.4% consistency in repeated testing, 100% precision in attribution at the function level for controlled cases, and robust performance in various types of application. The method reliably identifies *Phantom Rendering* patterns and successfully localizes root causes, validating its effectiveness for systematic performance analysis.

6 Threats to Validity and Limitations

6.1 Internal Validity

The precision of our *Phantom Rendering* detection is heavily dependent on the precision of the CPU instruction retirement measurements and the frame-level analysis. Although *CPU Retired Instructions* provides hardware-level precision, there might be some potential errors caused by system-level interference or hardware performance counters. In addition, since our evaluation only covers representative applications, we have not taken into account every case. The selected applications, while covering major categories (social media, e-commerce, entertainment), may not represent all possible UI interaction patterns or computational complexity levels. To address this limitation, we systematically varied the interaction parameters and selected applications with different complexity levels to maximize the coverage of the scenario. Finally, we only detect and localize the *Phantom Rendering* frame, but we don't have the ground truth for validating the accuracy of *Phantom Rendering* detection. We used quantitative metrics (CPU instruction retirement, frame flags) that provide objective measures of computational waste.

6.2 External Validity

Our evaluation was performed on a limited set of mobile devices running OpenHarmony. It is likely that they are not inclusive of all varieties of hardware and system versions in the mobile ecosystem. Different device capabilities, screen resolutions, and hardware performance characteristics could affect the detection accuracy of *Phantom Rendering*. In addition, our test scenarios cover common user interactions, but real-world usage patterns may involve more complex, unpredictable, or edge-case scenarios. The testing pattern may miss some possible user behaviors that could trigger *Phantom Rendering* issues. In addition, we note that the core concept of *HapPRDetection*—detecting computational waste by analyzing mismatches between pipeline workloads and actual visual output—is conceptually applicable to other mobile platforms that share the dual-thread rendering architecture, although our current implementation and evaluation are conducted on OpenHarmony. The architectural similarity across platforms provides a foundation for adaptation. Specifically,

applying this concept to Android would involve correlating UI thread `Choreographer.doFrame` calls with `RenderThread` draw command generation and `SurfaceFlinger` composition events, which are observable through Android's Perfetto tracing infrastructure. On iOS, the approach could be adapted to analyze the coordination between UIKit `CATransaction` operations and Core Animation's rendering pipeline through Instruments. The architectural similarity is further supported by the shared dependence on standard graphics libraries (e.g., Skia, OpenGL/Vulkan) across platforms. We conducted the evaluation on OpenHarmony strategically because the platform provides robust low-level framework access that enables precise hooking of frame rejection signals through its fully implemented composition manager subsystem, which performs a similar role to Android's `SurfaceFlinger`.

6.3 Limitations

Our approach to *Phantom Rendering* detection has some limitations. Our current framework treats *Phantom Rendering* as a binary problem (phantom vs. non-phantom), but in reality the boundary between legitimate UI computation and wasteful computation is not so clear. Some UI operations may be necessary to maintain the state of the application even if they do not produce immediate visual output. Furthermore, our approach focuses on run-time performance analysis and cannot detect *Phantom Rendering* issues embedded in the application architecture or design patterns. In addition, our implementation is tightly coupled with OpenHarmony's specific architecture and rendering pipeline, making adaptation to other platforms challenging in practice. Regarding performance overhead, the data collection phase relies on OpenHarmony's built-in HiPerf tool, and its overhead is determined by HiPerf's sampling configuration rather than *HapPRDetection* itself. To prevent probe effects that could distort the measurements, all heavy computational tasks such as differential analysis and hierarchical attribution are performed offline, ensuring that the mobile runtime remains representative during data collection. Nevertheless, the HiPerf sampling process inherently introduces minor CPU and memory overhead during data collection, which may affect measurement accuracy in performance-critical scenarios. Finally, while we systematically varied interaction parameters, our test scenarios may not capture all possible user interaction patterns, especially edge cases and complex multi-touch gestures. These limitations highlight the complexity of the *Phantom Rendering* problem and other performance problems. And more continued research and development should be done in this area. Each limitation represents an opportunity for future work to enhance the framework's capabilities and address broader performance analysis challenges.

7 Conclusion

This paper characterizes *Phantom Rendering*, a widespread but previously unexplored performance issue in terms of systematic detection and quantification, where mobile applications perform substantial UI computations that produce no visible output. Through empirical investigation of the top-22 OpenHarmony applications by download volume with 193 test steps, we demonstrate that *Phantom Rendering* occurs in 19 test cases across 8 applications, with severe cases wasting up to 40% of computational resources. We establish *Phantom Rendering* as a significant phenomenon driven by dynamic user-interaction patterns, particularly content scrolling, real-time updates, and interactive animations. Critical cases consume an average of 18.3% of CPU cycles without visual output, directly correlated with user-reported battery drain and overheating issues. To address this challenge, we introduce *CPU Retired Instructions* as a hardware-level metric to quantify computational waste, which can offer a fine-grained insight at the function level and develop *HapPRDetection*, a differential analysis framework for systematic detection of *Phantom Rendering*. Our approach achieves 94.4% detection consistency and 100% function-level attribution accuracy without requiring baseline comparisons. Our findings reveal *Phantom Rendering* as a fundamental

challenge in modern mobile UI architectures. As applications become increasingly interactive and data-driven, systematic detection and mitigation of such computational inefficiencies will be essential for sustainable mobile computing. Future work should investigate *Phantom Rendering* across platforms and develop predictive mitigation strategies.

8 Data Availability

Our code and results are available in https://github.com/SMAT-Lab/PhantomRendering.git.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 62572024, 62502021), by the National Key Research and Development Program of China (No. 2024YFB4506300), by "the Fundamental Research Funds for the Central Universities".

References

[1] Apple. [n. d.]. Instruments Tutorials. https://developer.apple.com/tutorials/instruments.

[2] John Austin. 2000. Performance analysis and performance diagnostics. *Handbook of applied behavior analysis* (2000), 321–349.

[3] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 641–660. doi:10.1145/2509136.2509549

[4] Abhishek Banerjee, Tanmay Mehta, Vijayalakshmi Srinivasan, and Massoud Pedram. 2007. Accurate microarchitecture-level power modeling on real processors. *IEEE Micro* 27, 6 (2007), 26–35.

[5] W Lloyd Bircher and Lizy K John. 2007. Complete system power estimation: A trickle-down approach based on performance events. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 158–168. doi:10.1109/ispass.2007.363746

[6] Susanne Braun, Frank Elberzhager, and Konstantin Holl. 2017. Automation support for mobile app quality assurance–a tool landscape. *Procedia Computer Science* 110 (2017), 117–124. doi:10.1016/j.procs.2017.06.129

[7] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *Proceedings of the 7th NASA Formal Methods Symposium*. Springer, 3–11. doi:10.1007/978-3-319-17524-9_1

[8] Aaron Carroll and Gernot Heiser. 2010. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. 21–21.

[9] Haonan Chen, Daihang Chen, Yizhuo Yang, Lingyun Xu, Liang Gao, Mingyi Zhou, Chunming Hu, and Li Li. 2025. ArkAnalyzer: The Static Analysis Framework for OpenHarmony. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 136–147. doi:10.1109/icse-seip66354.2025.00018

[10] Haonan Chen, Mingyi Zhou, Yanjie Zhao, and Li Li. 2026. HapFlow: The Taint Analysis Framework for OpenHarmony Apps. *ACM Transactions on Software Engineering and Methodology* (2026). doi:10.1145/3793863 Online publication date: February 2026.

[11] Luis Cruz and Rui Abreu. 2017. Performance-based guidelines for energy efficient mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 46–57. doi:10.1109/mobilesoft.2017.19

[12] Android Developers. 2023. Improve your code with lint checks. https://developer.android.com/studio/write/lint/ Android Studio static analysis tool.

[13] Android Developers. 2023. Power Rails: Android Power Profiling. https://source.android.com/devices/tech/power Android system-level power profiling tool.

[14] Dimitris Economou, Suzanne Rivoire, and Christos Kozyrakis. 2006. Full-system power analysis and modeling for server environments. *Workshop on Modeling, Benchmarking, and Simulation* (2006).

[15] OpenAtom Foundation. 2023. HiPerf Performance Profiler. https://gitee.com/openharmony/developtools_hiperf OpenHarmony performance monitoring and profiling tool.

[16] OpenAtom Foundation. 2023. Hypium Testing Framework. https://gitee.com/openharmony/testfwk_arkxtest OpenHarmony automated testing framework.

[17] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. 2017. Every pixel counts: Fine-grained UI rendering analysis for mobile applications. In *IEEE INFOCOM 2017-IEEE Conference on*

Computer Communications. IEEE, 1–9. doi:10.1109/infocom.2017.8057023

[18] WebAssembly Community Group. 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/ W3C WebAssembly specification for high-performance web applications.

[19] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review* 40, 3 (2012), 13–17. doi:10.1145/2425248.2425252

[20] David Ke Hong, Ashkan Nikravesh, Z Morley Mao, Mahesh Ketkar, and Michael Kishinevsky. 2019. Perfprobe: A systematic, cross-layer performance diagnosis framework for mobile platforms. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 50–61. doi:10.1109/mobilesoft.2019.00018

[21] Monsoon Solutions Inc. 2020. Monsoon Power Monitor. https://www.msoon.com/ Hardware-level mobile device power measurement.

[22] Canturk Isci and Margaret Martonosi. 2003. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. 93–104. doi:10.1109/micro.2003.1253186

[23] Jonathan G Koomey. 2007. Estimating total power consumption by servers in the US and the world. *Lawrence Berkeley National Laboratory* (2007).

[24] Inc. Kwai. 2025. AndroidGodEye - A performance monitor tool for Android applications. https://github.com/KwaiAppTeam/AndroidGodEye. Accessed: August 22, 2025.

[25] Gwangmin Lee, Seokjun Lee, Geonju Kim, Yonghun Choi, Rhan Ha, and Hojung Cha. 2019. Improving energy efficiency of android devices by preventing redundant frame generation. *IEEE transactions on mobile computing* 18, 4 (2019), 871–884. doi:10.1109/tmc.2018.2844202

[26] Xianfeng Li, Gengchao Li, and Xiaole Cui. 2020. Retriple: reduction of redundant rendering on android devices for performance and energy optimizations. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6. doi:10.1109/dac18072.2020.9218517

[27] Farong Liu, Mingyi Zhou, Yakun Zhang, Ting Su, Bo Sun, Jacques Klein, Xiang Gao, and Li Li. 2025. HapTest: The Dynamic Analysis Framework for OpenHarmony. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*. 422–431. doi:10.1145/3696630.3728565

[28] Wei Liu, Feng Lin, Linqiang Guo, Tse-Hsun Chen, and Ahmed E Hassan. 2025. GUIWatcher: Automatically Detecting GUI Lags by Analyzing Mobile Application Screencasts. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 46–56. doi:10.1109/icse-seip66354.2025.00010

[29] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024. doi:10.1145/2568225.2568229

[30] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. 2012. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *2012 41st International Conference on Parallel Processing*. IEEE, 48–57. doi:10.1109/icpp.2012.31

[31] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234. doi:10.1145/2491411.2491450

[32] Radhika Mittal, Aman Kansal, and Ranveer Chandra. 2012. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*. 317–328. doi:10.1145/2348543.2348583

[33] Ricardo R Pastrana-Vidal, Jean-Charles Gicquel, Catherine Colomes, and Hocine Cherifi. 2004. Frame dropping effects on user quality perception. *Proc. 5th Int. WIAMIS* (2004).

[34] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. 2012. AppInsight: mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. 107–120.

[35] Eduardo Salas, Michael A Rosen, C Shawn Burke, Denise Nicholson, and William R Howse. 2007. Markers for enhancing team cognition in complex environments: The power of team performance diagnosis. *Aviation, space, and environmental medicine* 78, 5 (2007), B77–B85.

[36] Aaron Shye, Benjamin Scholbrock, and Gokhan Memik. 2009. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 168–178. doi:10.1145/1669112.1669135

[37] Linhai Song and Shan Lu. 2017. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 370–380. doi:10.1109/icse.2017.41

[38] Android Studio. [n. d.]. Perfetto - System profiling, app tracing and trace analysis. https://developer.android.google.cn/tools/perfetto?hl=zh-cn.

[39] Gleb Tarasov and Booking.com. 2024. PerformanceSuite: A Swift-based iOS library for monitoring app performance and quality metrics. https://github.com/bookingcom/perfsuite-ios iOS performance monitoring framework.
[40] Vincent M Weaver. 2013. Linux perf_event features and overhead. In *Proceedings of the 2nd International Workshop on Performance Analysis of Workloads on Modern Architectures*. ACM.
[41] Claas Wilke, Sebastian Richly, Sebastian Götz, Christian Piechnick, and Uwe Aßmann. 2013. Energy consumption and efficiency in mobile applications: A user feedback study. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. IEEE, 134–141. doi:10.1109/greencom-ithings-cpscom.2013.45
[42] Jing Xu. 2008. Rule-based automatic software performance diagnosis and improvement. In *Proceedings of the 7th international workshop on Software and performance*. 1–12. doi:10.1145/1383559.1383561
[43] Yu Yan, Songtao He, Yunxin Liu, and Longbo Huang. 2015. Optimizing power consumption of mobile games. In *Proceedings of the Workshop on Power-Aware Computing and Systems*. 21–25. doi:10.1145/2818613.2818746
[44] Yizhuo Yang, Lingyun Xu, Mingyi Zhou, and Li Li. 2025. Context-Sensitive Pointer Analysis for ArkTS. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 3262–3273. doi:10.1109/ASE63991.2025.00269
[45] Feng Zheng, Turner Whitted, Anselmo Lastra, Peter Lincoln, Andrei State, Andrew Maimone, and Henry Fuchs. 2014. Minimizing latency for augmented reality displays: Frames considered harmful. In *2014 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE, 195–200. doi:10.1109/ismar.2014.6948427

Received 2025-09-12; accepted 2025-12-22