Dissecting and Streamlining the Interactive Loop of Mobile Cloud Gaming

Yang Li[1*], Jiaxing Qiu[1*], Hongyi Wang[1], Zhenhua Li[1], Feng Qian[2], Jing Yang[1]
Hao Lin[1,4], Yunhao Liu[1], Bo Xiao[3], Xiaokang Qin[3], Tianyin Xu[4]
[1]*Tsinghua University* [2]*University of Southern California* [3]*Ant Group* [4]*UIUC*

Abstract

With cloud-side computing and rendering, mobile cloud gaming (MCG) is expected to deliver high-quality gaming experiences to budget mobile devices. However, our measurement on representative MCG platforms reveals that even under good network conditions, all platforms exhibit high interactive latency of 112–403 *ms*, from a user-input action to its display response, that critically affects users' quality of experience. Moreover, jitters in network latency often lead to significant fluctuations in interactive latency.

In this work, we collaborate with a commercial MCG platform to conduct the first in-depth analysis on the interactive latency of cloud gaming. We identify VSync, the synchronization primitive of Android graphics pipeline, to be a key contributor to the excessive interactive latency; as many as five VSync events are intricately invoked, which serialize the complex graphics processing logic on both the client and cloud sides. To address this, we design an end-to-end VSync regulator, dubbed LoopTailor, which minimizes VSync events by decoupling game rendering from the lengthy cloud-side graphics pipeline and coordinating cloud game rendering directly with the client. We implement LoopTailor on the collaborated platform and commodity Android devices, reducing the interactive latency (by ~34%) to stably below 100 *ms*.

1 Introduction

Recent years have witnessed the emergence and prosperity of cloud gaming for mobile games (termed *mobile cloud gaming*, or MCG), which enables high-quality gaming experiences on mid- to low-end devices. Typically, mobile games run in virtualized environments (e.g., an emulator/a container) hosted on cloud servers, and rendered frames are live-streamed over the network to a thin client app on the user device.

Motivation. To achieve smooth gaming experiences, MCG platforms are expected to provide users with a high media streaming bandwidth (e.g., 10+ Mbps) [23] and a low *interactive latency* (i.e., the delay between a user-input action and when the game's response to that action manifests at the client) within ~100 *ms* [54]. In practice, the former has been largely fulfilled by the deployment of geo-distributed cloud/edge servers [24]. Nevertheless, the latter, as a key metric that directly determines the QoE of MCG, seems far from satisfactory [3, 54]. MCG's interactive latency has not been well understood—it is more than network latency due to the complexity of cloud-side processing and virtualization [72].

We conduct the first in-depth comparative measurement study of MCG and CCG (cloud gaming for console games, or *console cloud gaming*) on interactive latency characteristics in practice. Our measurement is based on eight representative cloud gaming platforms: GeForce Now [61], Boosteroid [20], JoyArk [41], CloudMoon [26], NetEase Cloud Gaming [58], Tencent Pioneer [78], Tencent Start [79], and X-MCG (anonymized due to their commercial concerns), regarding both mobile and console games under diverse scenarios. The study reveals important and surprising results:

- Our measured minimum interactive latency of MCG is as high as 112 *ms*, 42% larger than that of CCG. This implies that the processing logic of MCG should be more complex than that of CCG and the interactive latency of today's MCG cannot meet the users' requirements (<100 *ms*).
- The interactive latency has little to do with the user request workload but much to do with the game rendering workload. Specifically, the interactive latency looks similar during peak and non-peak hours (indicating sufficient scalability), but surprisingly (up to 48%) higher for simple 2D games than for complex 3D games (mostly relating to the unbalanced resource allocation strategy of a platform).
- On average, network latency only accounts for 17% of the interactive latency, indicating that network is not the key bottleneck. In fact, a jitter (≤2 *ms*) in network latency often has a "butterfly effect" on the interactive latency (≥10 *ms*).

The above drive us to go beyond RTT and understand the *interactive loop* of MCG, including every step in the back-and-forth MCG processing pipeline. However, the interactive loop of commercial MCG platforms is rarely studied in depth as it involves not only network links but also complex backend processings in the cloud, which are typically invisible.

Analysis. To deeply understand the interactive loop of MCG, we contacted developers of each platform. One platform X-MCG (anonymized due to their commercial concerns) shared their design and agreed to collaborate with us on improving interactive latency. Its cloud side relies on a state-of-the-art Android emulator *Trinity* [33] (optimized for high-performance graphics rendering) and a widely-used streaming tool *Sunshine* [51]. Its client side is extended from *Moonlight* (the client of Sunshine), supporting hardware-accelerated video decoding and High Dynamic Range (HDR) streaming. We

* Co-primary authors. Zhenhua Li is the corresponding author.

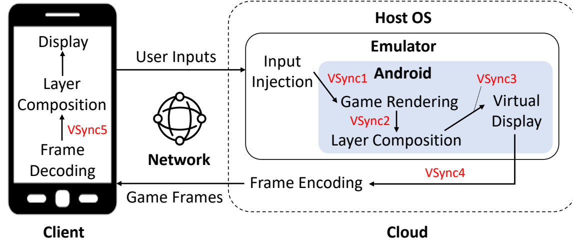


Figure 1: The interactive loop of X-MCG.

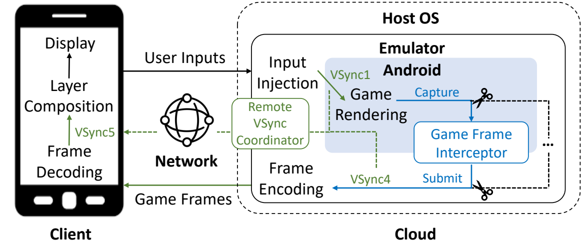


Figure 2: The interactive loop of X-MCG using LoopTailor.

take this unique opportunity to first expose to the public a state-of-the-art MCG system's interactive loop, and uncover the root cause of the unsatisfactory interactive latency.

To precisely trace the data flow of a game frame in X-MCG, we instrument relevant system services of Android, the GPU module of *Trinity*, and core modules of *Sunshine/Moonlight*. We find that the key bottleneck of the interactive latency is the cloud-side graphics pipeline. More in depth, the major latency of the cloud-side graphics pipeline does not lie in game rendering (23%), but the Vertical Synchronization (VSync) operations [2, 18, 53] (36%) in mobile OSes. As a synchronization primitive, VSync synchronizes frame production and consumption rates (e.g., the output frame rate of the GPU and the refresh rate of the display) among different stages in a graphics pipeline to avoid *screen tearing* [53] (i.e., a visual artefact that occurs when a display device shows pixels from multiple frames in a single screen draw). Given that graphics pipelines of mobile systems are designed to handle diverse workloads with power-constrained hardware and fixed-refresh-rate display [18], VSync is built into the Android graphics pipeline as a system-level always-on mechanism.

Given the multi-component and multi-stage nature of the MCG graphics pipeline, severe performance penalty is incurred as each VSync event may cause an uncertain latency. As shown in Figure 1, a game frame encounters as many as five VSync events, creating up to 83 *ms* extra latency. Given the periodicity of VSync events, even a slight network jitter (e.g., 1 *ms*) can lead to a frame deadline miss, resulting in a significant fluctuation (e.g., 13 *ms*) in interactive latency. Note that the tight coupling of VSync with Android makes it not limited to specific emulators or streaming tools, but affects other representative MCG solutions (e.g., DroidCloud [47]) as well, incurring even higher (20%+) latencies (§6).

We argue that the VSync overhead in the MCG graphics pipeline can be effectively optimized away with specialized system design. Basically, current uses of VSync are built for general-purpose mobile usage, where multiple apps and system services may render simultaneously and thus require sophisticated synchronizations to achieve smooth and correct rendering. However, in MCG, the game app process monopolizes the rendering pipeline, and thus game-irrelevant synchronizations within the cloud (i.e., VSync2 and VSync3) could be avoidable, given that layer composition and virtual display only append system UIs (e.g., a status bar) to a game frame. Essentially, only coordination with the client is a necessity, leaving considerable space for optimization inside the cloud.

Optimization. We present LoopTailor, an end-to-end VSync regulator to decouple game rendering from the lengthy cloud pipeline (thus reducing four cloud VSync events to two), and coordinate its pace directly with the client (VSync5). To do so, LoopTailor carefully streamlines MCG's interactive loop. As shown in Figure 2, LoopTailor consists of two modules: *Game Frame Interceptor* which extracts and reroutes game frames to bypass unnecessary VSyncs (VSync2 and VSync3), and *Remote VSync Coordinator* which coordinates ahead of time the remaining VSyncs (VSync1, VSync4, and VSync5) on both sides. LoopTailor's approach is transparent to specific VSync implementations, making it a generic approach requiring minimum modifications to the host/guest OS.

Nevertheless, realizing LoopTailor is challenging. First, extracting game frames halfway from the pipeline can introduce non-trivial guest-host frame copy overheads. Second, VSync coordination between the cloud and the client requires accurately predicting latencies of intervening stages, as a prediction error may cause a frame deadline miss at the client.

For intercepting game frames, conventional approaches like app-level render redirection [15, 87] and framework-level buffer hooking [83] induce substantial overheads of frame copies in the virtual mobile graphics stack. Our solution instead breaks the boundary of the guest system and the host virtual GPU to efficiently identify render instructions issued by the game and capture the game frames *in place*, thus minimizing frame copies between the CPU and the GPU. The Interceptor then immediately forwards the game frames to the encoder, so that VSync2 and VSync3 are bypassed.

For multi-stage latency prediction, we discover that the latencies (incurred by game rendering, frame encoding, network transmission, and frame decoding), despite heterogeneous and fluctuating, can be forecast hierarchically with a low standard deviation (<1.3 *ms*) based on their latent correlations. Thus, we devise a Synergetic VSync Alignment approach which proactively adjusts the execution time of rendering/encoding operations across the cloud-side pipeline, so as to minimize the client-side frame deadline miss rate (0.3%).

We implement LoopTailor atop X-MCG with 15K lines of C/C++/Python code and evaluate its real-world performance with 100 mobile devices. Our prototype is applicable to both single- and multi-player mobile games. The results

demonstrate that LoopTailor can achieve desirable interactive latencies of 82-96 *ms* (with X-MCG's geo-distributed edge servers), 34% lower than that of X-MCG's original solution. The average ratio of interactive latency fluctuation to network jitter is greatly reduced from 11.52 to merely 1.23.

Contribution. We make the following contributions:

- We make a month-long measurement study on 8 representative cloud gaming platforms, revealing important results on interactive latency and optimization opportunities.
- We conduct the first in-depth study of MCG's interactive loop. Our analysis identifies the VSync mechanism in the cloud-side graphics pipeline of MCG as a key contributor to the excessive interactive latency, which can be effectively optimized with specialized design for MCG.
- We design and implement LoopTailor to streamline X-MCG's interactive loop, achieving sub-100 *ms* interactive latency in real-world deployments. The code and data are released at `https://MCGlatency.github.io`.

2 Measurement Study in the Wild

In this section, we introduce our methodology of measuring the interactive latency of commercial cloud gaming platforms (§2.1), and present multifold data analysis results (§2.2).

2.1 Measurement Methodology

Due to commercial interests and security concerns, almost all the commercial cloud gaming platforms are closed-source systems and provide very limited technical documentation to the public. Therefore, we measure the interactive latency of a platform from the outside in a record-and-recognize approach.

First, we use high-speed cameras to record videos of user interactions with games with an ultra-high frame rate of 2000 FPS (Frame Per Second) and Full-HD resolution (1920×1080). We position 2 cameras in front of and at the side of the device screen respectively, so that both user actions and visual game contents can be clearly observed. For recorded videos, we examine each frame in a hybrid (manual and algorithmic) approach to recognize the timestamp when the user input action happens, as well as the timestamp when the corresponding response of the action manifests.

In detail, we record specific interaction patterns for each game (e.g., invoking game menus, changing views, and touching in-game buttons), which are performed with styluses instead of fingers to ease detection. After recording, we leverage conventional computer vision algorithms [43, 90, 92] to detect and track objects in the frames. We first detect the stylus and the screen from the side view of the device screen, and then recognize the very frame (when the distance between the stylus and the screen reaches the minimum) as the start of an interaction. After that, we track the pattern-specific response objects (e.g., button highlight) case-by-case in the front view of the device screen to identify the end of an interaction. Such an approach is common practice in existing work for collecting visual-related delays [31], and the accuracy of the algorithms can reach 99+% according to our random sampling-based manual checking. In this way, the interactive latency can be deemed as the time interval between the two timestamps, with a negligible error of up to two frames (∼1.0 *ms*) introduced by the camera.

Table 1: Eight representative cloud gaming platforms studied in this work, in alphabetical order by their names.

ID	Platform	Country	Type
1	Boosteroid	USA	CCG
2	CloudMoon	USA	MCG
3	GeForce Now	USA	CCG
4	JoyArk	USA	MCG+CCG
5	NetEase Cloud Gaming	China	MCG+CCG
6	Tencent Pioneer	China	MCG
7	Tencent Start	China	CCG
8	X-MCG	China	MCG

Further, we are curious about the network impact on the interactive latency since this is what common users are the most concerned with. Once a user suffers from high interactive latency, s/he is most likely to complain about the network latency. To address the concern, almost all the commercial cloud gaming platforms have on-screen widgets presenting the real-time network latency (via their client apps), which greatly facilitates our recording the network latency.[1] Since a low network bandwidth can also lead to high interactive latency, we periodically record the downlink bandwidth with the lightweight Android API *NetworkStatsManager.querySummaryForDevice()* in the meantime.

As listed in Table 1, we study 8 representative cloud gaming platforms that are highly popular in either the USA or China, using 100 mobile devices of 43 device models (1.8-3.3 GHz CPU frequency, 4-16 GB memory, and Android 9-13) geo-distributed in the corresponding countries (50 in 3 cities in the USA and 50 in 3 cities in China). Among them, three are CCG platforms, three are MCG platforms, and two support both CCG and MCG. We include CCG in our measurements to understand latency differences between MCG and CCG.

For each platform, we select its top-10 popular games to measure the interactive latencies with different radio access technologies (RATs): 5G (37%), 4G (29%), and WiFi (34%). We set the streaming resolution and frame rate of each platform to 1080P and 60 FPS respectively, which are the default settings of most platforms. Also, we use the 100 mobile devices to play games at different times (0:00, 3:00, ..., 21:00, UTC+8) every day during a whole month (May. 2023), so as to figure out the influence of user request workloads. The duration of game playing each time is five minutes.

Ethical Claim. All the measurements and analyses throughout this work are conducted with explicit permission of users and no ethical concern is raised in this work.

[1] Some platforms may not be reporting accurate latency values, and different platforms may have different methodologies to calculate the latency.

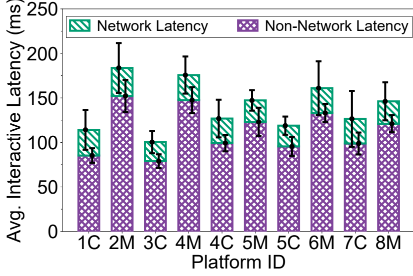


Figure 3: Average interactive latency of each platform. "M" and "C" denote MCG and CCG respectively.

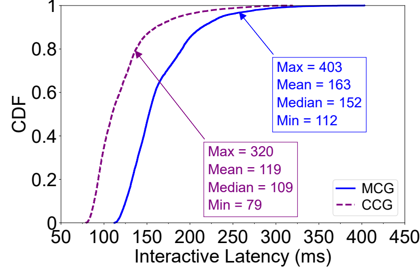


Figure 4: Interactive latencies of our studied MCG and CCG platforms.

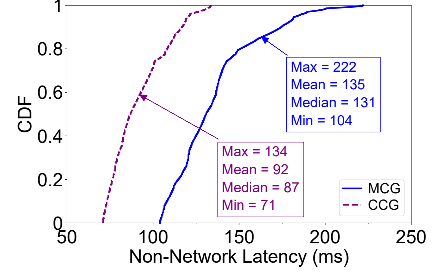


Figure 5: Non-network latencies (i.e., interactive latency excluding network latency) of MCG and CCG platforms.

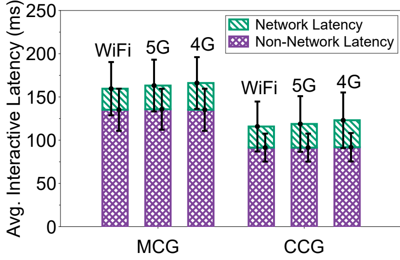


Figure 6: Average interactive latencies of our studied MCG and CCG platforms with different RATs.

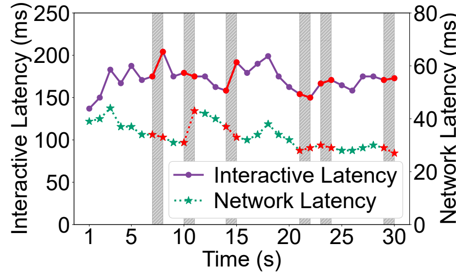


Figure 7: Interactive and network latencies during a typical measurement on top of Tecent Pioneer.

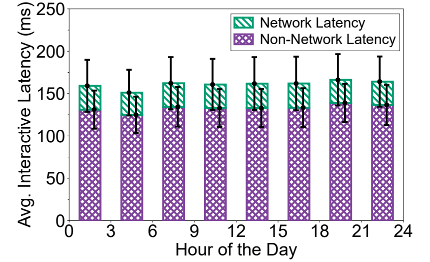


Figure 8: Average interactive latency of Tencent Pioneer at different times of a typical day (time zone: UTC+8).

2.2 Measurement Results

Among our records of individual measurements, a small portion (6%) are undesirable because it is difficult for us to analyze their relevant interactive latencies. For example, when the downlink bandwidth is insufficient, we can hardly quantify its remarkable and highly fluctuating influence on the interactive latency, thus making our analyses obscure. After ruling out such cases, we obtain a total of 20,096 valid records, which reveal quite a few surprising results with regard to interactive latencies of today's commercial MCG platforms.

General Statistics. All the studied MCG platforms suffer from unsatisfactory interactive latencies ranging from 112 *ms* to 403 *ms*—recall in §1 that smooth gaming experiences require a low interactive latency within 100 *ms*. In contrast, in non-negligible (35%) cases interactive latencies of CCG stay below 100 *ms*. Obviously shown in Figure 3 (in all the bar charts throughout this paper, each error bar denotes the standard deviation of the corresponding sub-bar), interactive latencies of MCG are generally higher than those of CCG.

As shown in Figure 4, our measured minimum interactive latency of MCG (112 *ms*) is 42% larger than that of CCG (79 *ms*), and the average interactive latency of MCG (163 *ms*) is 37% larger than that of CCG (119 *ms*). Regarding average interactive latency, even the fastest MCG platform (8M: X-MCG) is slower than the slowest CCG platform (4C: JoyArk). We attribute the above to the more complex processing logic of MCG compared with CCG, i.e., the mobile OS virtualization specially required by MCG. In fact, CCG platforms typically use hardware-assisted virtualization techniques (e.g., SR-IOV [62]) to share resources to VM instances, whose performance is close to direct hardware access. But for MCG, given the poor driver support of desktop GPUs in mobile OSes, para-virtualized GPUs with additional virtualization logic are usually required. Hence, there exists a gap in rendering performance between the CCG and MCG backends.

Network Impact. Contrary to most users' common beliefs, most of the excessive MCG latency is not attributed to the network. With the network latency excluded (then the remaining latency is referred to as the *non-network latency*), the minimum non-network latency of MCG is still as high as 104 *ms* (>100 *ms*), 46% larger than that of CCG (71 *ms*), as indicated in Figure 5. This is because all our studied platforms have deployed geo-distributed cloud and/or edge servers across either the USA or China, making the network latency pretty low in most cases. Concretely, in 81% of the valid records, the user device lies in the same city as the cloud/edge server. More in detail, no matter which RAT is used, the network latency (averaging between 24 *ms* and 31 *ms*) only takes up a small portion (averaging between 15% and 25%) of the interactive latency, as indicated in Figure 6.

Common wisdom suggests a positive correlation between interactive latency and network latency. Surprisingly, in a non-trivial portion (13%) of our measurements the interactive latency is negatively correlated with the network latency. To visualize this, Figure 7 depicts results during a typical measurement that includes such unexpected cases. Dots under

shadows denote cases of a negative correlation between interactive latency and network latency. The reason behind the above counter-intuitive phenomenon will be detailed in §3.3.

User Request Workload. Figure 8 presents average interactive latencies at different times of a typical day on Tencent Pioneer, the most popular MCG platform in our measurement. There is no significant difference between the interactive latencies during peak hours (12:00 – 21:00) and non-peak hours (00:00 – 12:00, 21:00 – 24:00). Further, we do not see notable variation among the interactive latencies in different days of a week, either. The above hold true for other MCG platforms as well, revealing that today's MCG platforms possess sufficient scalability to serve a fluctuating user request workload.

Game Rendering Workload. Recall that for each platform, we select its top-10 popular games to play and measure interactive latencies, including simple 2D games, complex 3D games, and so forth. Surprisingly, we note that on all MCG platforms, simple 2D games with light rendering workloads suffer from higher interactive latencies (10% on average) compared with graphics-intensive 3D games. Given that official documents of some MCG platforms mention that they selectively schedule GPU resources for different game types [80], we suspect that the surprising observation might be owing to unbalanced resource allocation strategies, i.e., overly prioritizing the QoE of complex 3D games while ignoring requirements of lightweight 2D games. To validate the conjecture, we consulted the customer service of each MCG platform, and two of them acknowledge that simple 2D games are oftentimes subject to insufficient GPU resource allocations.

3 Diagnosing the Undesirable Interactive Latency of MCG

To further discover the root causes behind the above observations, we contacted the developers of the five commercial MCG platforms. Eventually, the developers of X-MCG agreed to share their design with us and collaborate with us to diagnose the undesirable interactive latency atop their platform.

In this section, we first introduce important background knowledge regarding mobile graphics pipelines and VSync (§3.1). Then, we dissect the end-to-end interactive loop of X-MCG into 16 pipeline stages (§3.2). Afterwards, we measure the latencies of key stages in the X-MCG pipeline to quantitatively explain why the interactive latency is unsatisfactory (§3.3). We will discuss system architectures of other MCG solutions and the generalizability of our findings in §6.

3.1 Mobile Graphics Pipeline and VSync

The graphics pipelines of modern mobile OSes consist of multiple *stages* due to the users' diverse needs for graphics; each stage performs its unique functionality. When a game is played on an Android device, for instance, the game is typically the beginning of the graphics pipeline. The game performs graphics rendering by writing its contents into a frame buffer (or *frame* for short) from a buffer queue termed `Surface` [9]. At the same time, other system apps (e.g., system UI) may render contents into their own `Surfaces`.

Various `Surfaces` are connected to a compositor called `SurfaceFlinger` [10], whose job is to merge the contents of individual `Surface` layers into one or more graphic frames for display. In other words, after game frames are rendered, `SurfaceFlinger` conducts *Layer Composition* [14] on the frames. The composited result is then sent to a Hardware Composer (`HWcomposer`) module [13], which submits the contents to the actual display hardware.

It is worth noting that the stages usually work in their own paces, which can cause frame access issues. For example, while `SurfaceFlinger` regularly conducts Layer Composition at the refresh rate of the display, the pace of game rendering can be rather uncertain. When the game delivers a new frame to `SurfaceFlinger` while it is compositing, the composited result may contain portions of multiple game frames, leading to the *screen tearing* [53] problem.

To address screen tearing, modern mobile OSes typically adopt the VSync mechanism; other mechanisms (e.g., Triple Buffering [34]) have also been proposed, but each of them alone is generally unsuited to power-restrained mobile systems without the help of VSync [2]. As mentioned before, VSync is a classic access control mechanism (or synchronization primitive) for the graphics pipeline, which controls the frame intervals of two adjacent stages to make the frame access exclusive. It works in an event-driven manner: frame consumption is triggered periodically by VSync events at a pace specified by the consumer (e.g., when the display with a refresh rate of 60 FPS is the consumer, the period is 1/60 s). Note that the frequency and period of VSync events are determined by the frame rate of the consumer in a graphics pipeline, and thus cannot be adjusted freely.

In each period, when the producer finishes frame production, it waits for the *vertical blanking interval* [44] (i.e., the time interval between the end of a production and the next VSync event of the consumer). Then, it delivers the frame to the consumer only when no consumption (e.g., frame drawing) is in progress, so as to avoid screen tearing while saving its power (i.e., preventing the producer from generating too many frames that cannot be processed by the consumer). In other words, when the processing of one frame is finished, the graphics pipeline will not process a new frame immediately; instead, the processing of a new frame will be delayed till the signaling of the next VSync event, thus creating an uncertain latency. Worse still, the latencies can accumulate in a lengthy graphics pipeline, leading to unsatisfactory user experiences.

3.2 The Interactive Loop of X-MCG

X-MCG is developed and commercialized through the combinatory use of Trinity and Sunshine on the cloud side, as well as extending Moonlight (the client of Sunshine) as the client app. The developers of X-MCG carefully make the choice by conducting benchmarks on various existing MCG solu-

tions, which will be elaborated in §6. Further, they customize the AOSP (Android Open Source Project) [11] system that runs within the cloud servers by pruning unnecessary system services like `BatteryService` [76].

The high-level interactive loop of X-MCG is presented in Figure 1. It comprises as many as 16 stages as shown in Figure 9, which will be detailed below. The pipeline starts from the user interaction with the client display hardware, and the client app captures user inputs **(Stage 1: User Input Capture)**. Inputs are then sent to the cloud-side streaming component Sunshine **(Stage 2: User Input Transmission)**. Upon receiving inputs, Sunshine injects them into the mobile game application that runs on the guest OS **(Stage 3: Input Injection)**. As mentioned before, the guest OS is a customized AOSP system that runs within the Trinity emulator.

Afterwards, the game prepares to process the inputs and render the game graphics. To avoid screen tearing and conserve power, mobile games synchronize the game's frame rate to a portion of the display refresh rate of the device [28, 81]. As explained in §3.1, they use VSync events to control how quickly game frames should be produced, which is usually specified by the developer/user via an in-game option. Consequently, user inputs need to wait for a VSync event before it gets processed and the game rendering starts **(Stage 4: VSync1)**, and VSync1 cannot be disabled as it is built deep into the game rendering logic for supporting fixed-refresh-rate mobile display.

With regard to game rendering, modern mobile games usually use rendering APIs like OpenGL ES [16] to render into their `Surfaces` **(Stage 5: Game Rendering)**. In order to achieve GPU-accelerated rendering in AOSP, typical solutions (e.g., Google Android Emulator [8] and DroidCloud [47]) forward render instructions from the GPU driver in AOSP to the host-side virtual GPU device using remote procedure calls (RPCs), which turns out to be slow and expensive. In contrast, through the novel *graphics projection* mechanism [33], the Trinity emulator directly resolves the vast majority (>99%) of rendering API calls within the guest GPU driver, significantly reducing RPC calls and improving the rendering performance.

When game rendering finishes, the results are submitted to `SurfaceFlinger`. `SurfaceFlinger` has to work in the same pace with the display hardware, and therefore needs to wait for VSync **(Stage 6: VSync2)** before compositing the output of the **game app** including system UIs **(Stage 7: Layer Composition I)**. Subsequently, the composited frame is sent to `HWcomposer`, which waits until the display hardware has finished the scan-out [32] of the previous frame **(Stage 8: VSync3)**. Then, `HWcomposer` presents the frame to the virtual offscreen display device provided by the Trinity emulator **(Stage 9: Virtual Display)**. VSync2 and VSync3 can introduce fluctuating delays due to highly variable game rendering workloads as well as possible resource contention in commercial MCG platforms, given that hardware resources are typically shared to a large number of virtualized instances

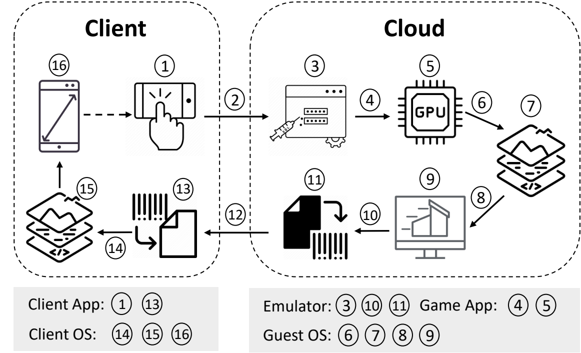


Figure 9: The 16 stages in the interactive loop of X-MCG.

with diverse game workloads.

Afterwards, Virtual Display forwards the frame to the Sunshine encoder. The encoder performs frame encoding according to the video frame rate set by the user, so a separate VSync is used **(Stage 10: VSync4)**. VSync4 exists because the client-side video frame rate may be different from the cloud-side game frame rate, and simple strategies like delivering frames immediately cannot satisfy the frame rate requirement of the client. The encoder and the Virtual Display are coordinated in the host virtual GPU, so that frame forwarding and encoding take place in the GPU with minimal frame copies between GPU and CPU **(Stage 11: Frame Encoding)**.

Over the network, the encoded game frame is transmitted to the client **(Stage 12: Game Frame Transmission)**, which collects and decodes the frame **(Stage 13: Frame Decoding)**. Then, similar to VSync2, the decoded frame waits for another VSync event triggered by the client `SurfaceFlinger` **(Stage 14: VSync5)** before conducting Layer Composition **(Stage 15: Layer Composition II)**. Note that VSync5, as a client-side VSync, can be disabled, but only with developer/root privileges. Furthermore, once it is disabled, the effect is global and other apps will also suffer from side-effects like screen tearing and increased power consumption, which is thus unacceptable to normal users. Finally, the composited frame is scanned out by the display hardware **(Stage 16: Display)**[2].

3.3 Root Cause Analysis

Among the 16 stages, we want to uncover which ones contribute the most to the interactive latency. To this end, we conduct an in-depth measurement on the end-to-end loop of X-MCG using 10 phones (with different device models) whose OSes are tailored for detailed latency recording. Given that delays of Stage 1 and Stage 16 are mostly induced by the display hardware and thus can hardly be measured by software means, we focus on measuring delays of Stages 2-15 (so that we can infer the sum delay of Stage 1 and Stage 16).

To precisely trace the data flow of a specific game frame in a lightweight manner, we measure passively by instru-

[2] Similar to VSync3, the composited frame needs to wait for another VSync event before the display scan-out, but as this VSync lies deep inside the client system and is invisible to user apps, we merge it to **Stage 16**.

Table 2: Latency breakdown of X-MCG (in unit of *ms*).

Category	Mean	Median	Min	Max
Network	15.7	16.4	7.9	24.1
VSync	43.9	44.0	4.5	82.9
Game Rendering	27.9	28.2	25.7	33.2
Video Processing	27.2	28.9	18.1	46.7
Layer Composition	5.4	5.1	3.0	6.6
Others	2.6	2.3	1.6	3.9

menting relevant system (Android, AOSP, Trinity and Sunshine/Moonlight) modules and recording the timestamps of relevant network packets. Specifically, we measure the network latency (Stage 2 and Stage 12) by piggybacking timestamps with network packets between Sunshine and Moonlight. In the cloud-side AOSP system, we instrument system services including `InputDispatcher`, `SurfaceFlinger`, and `HWComposer` in the framework layer, as well as the OpenGL ES library in the Hardware Abstraction Layer (HAL). In this way, delays of Stages 3-8 can be accurately recorded. In Trinity, we instrument the virtual display module to obtain the delay of Stage 9. Further, through tracking the encoding process of Sunshine, delays of Stage 10 and Stage 11 can be clearly observed. Finally, we instrument Sunshine and the client Android system to measure the delays of Stages 13-15.

Based on the above fine-grained instrumentation throughout the pipeline, we collected data at various times of each day during a week, and obtained the latency traces for a total of 35 hours of game playing. The measurement overhead is confined to a negligible level: the average CPU utilization is 1%, and the average memory consumption is 703 KB. We classify the delays of the measurable stages (Stages 2-15) in the loop into six categories, including Network (Stage 2 and 12), VSync (Stages 4, 6, 8, 10, and 14), Game Rendering (Stage 5), Layer Composition (Stages 7 and 15), Video Processing (Stages 11 and 13), and Others (Stages 3 and 9).

As listed in Table 2, in terms of mean latency, the VSync mechanism (5 VSync events) contributes the most (35.7%) to the non-network latency, even more than Game Rendering (22.7%) and Video Processing (22.1%). Moreover, delays incurred by VSync events are highly fluctuating between 4.5 *ms* and 82.9 *ms* (even more fluctuating than the network latency in our measurements), causing the interactive latency to vary significantly even when the network latency is stable. The above findings explain the counter-intuitive phenomenon we observe in §2.2—a lower network latency does not necessarily translate to a better interactive latency, since network is usually not the deciding factor. As for the “butterfly effect” mentioned in §1, it happens mostly when network jitters cause subsequent pipeline stages to miss VSync events according to our observation.

As shown in Table 3, each VSync event is required by an individual stage. Delving deeper into the loop, we discover that 5 VSync events are tightly coupled with other stages. VSync1 has been built into modern game engines like Unity [82] and Unreal [29] (Stage 5). VSync2, VSync3, and VSync5 are required by core graphics modules of both cloud-side and client-side Android systems, (namely `SurfaceFlinger` and `HWComposer`) (Stages 7, 9, 15). VSync4 is necessitated by the video encoder to encode videos at user-defined frame rates (Stage 11). Unfortunately, a lack of coordination between the VSync events causes an uncertain delay for each game frame with respect to each event. According to our measurements, the latencies of 5 VSync events all exhibit a nearly uniform distribution between 0 and 16.7 *ms*.

Table 3: Specific roles of the five VSync events and the average latency incurred by each VSync event.

Stage	Caused By	Location	Latency
4	Game Rendering	Game	8.5 *ms*
6	Layer Composition I	Guest Android	9.0 *ms*
8	Virtual Display	Guest Android	9.5 *ms*
10	Frame Encoding	Encoder	9.1 *ms*
14	Layer Composition II	Client Android	7.9 *ms*

Fortunately, while all the VSync events are *structurally* required and are hard to disable from the perspective of an MCG platform, not every VSync event is *functionally* useful. In the context of MCG, games are usually the only foreground apps besides system UIs (e.g., Android task bar), so Layer Composition and Virtual Display (Stage 7 and Stage 9) in the guest Android system (AOSP) only append game-irrelevant system UIs to a frame in almost all cases. Thus, these stages (along with their VSync events) are *functionally* unnecessary for MCG, as the game frames are ready to be displayed on the client after Game Rendering (Stage 5). This motivates our attempt to bypass some VSync events and decouple game rendering from the lengthy graphics pipeline, hoping to avoid or reduce their impact on the interactive latency.

4 System Design

Since the VSync mechanism is the biggest contributor to the interactive latency of X-MCG, we wish to minimize its negative impact. Guided by our diagnosis insights in §3.3, we design LoopTailor, an adaptive whole-loop regulator of VSync events for lowering the interactive latency of X-MCG to a desirable level (<100 *ms*). In this section, we first present a brief overview of LoopTailor along with the design goals (§4.1). Then, we detail its two key components: Game Frame Interceptor (§4.2) and Remote VSync Coordinator (§4.3).

4.1 LoopTailor Overview

LoopTailor sets out to meet the following design goals:

- Bypassing VSync events that are functionally unnecessary without disrupting the frame access control of the graphics pipeline (i.e., without harming the game’s visual display);
- Accurately predicting and aligning the remaining VSync events to minimize their incurred latency;
- Introducing very little computational overhead and negligible impact on the smoothness of game graphics.

Figure 2 shows the architectural overview of LoopTailor, which streamlines the interactive loop of X-MCG by cropping the cloud-side graphics pipeline and stitching it to the client-side pipeline in a fine-grained manner. At the heart of LoopTailor are two key components: 1) Game Frame Interceptor (GFI, §4.2) that efficiently captures the game frames ahead of schedule to bypass 2 (out of 5) VSync events; and 2) Remote VSync Coordinator (RVC, §4.3) that forecasts and synchronizes the remaining 3 VSync events by carefully coordinating the cloud and the client. Note that the five VSync events are necessitated by the Android system design and the nature of constant-frame-rate video codec; apart from our emulator-based solution, the high-level design of LoopTailor is applicable to other (potential) solutions like container-based or function-based serverless gaming as well.

4.2 Game Frame Interceptor (GFI)

GFI aims to capture raw frames from game rendering ahead of time, before any form of layer composition. In Android, the desired interception should take place between the game app and `SurfaceFlinger` (before Stage 6 in §3.2), and there exist multiple intuitive methods. App-level render redirection [17], for instance, redirects the app rendering `Surface` to a virtual display with hooking libraries such as XPosed [87], and uses the `MediaProjection` API [15] to directly record rendered contents. Framework-level buffer hooking [83] adds a hook inside `Surface`, and performs interceptions whenever a game-owned `Surface` delivers a frame to `SurfaceFlinger`.

Nevertheless, for MCG where captured frames need to be subsequently encoded and sent to the client device, such intuitive approaches incur considerable frame copy overheads (note that frame encoding in the original X-MCG pipeline is in-place, c.f., Stage 11 in §3.2). In essence, both approaches capture frames solely from *inside* the guest, but ultimately send the frames using the network provided by the host. Therefore, data copies between the guest and host are unavoidable.

To realize low-overhead frame interception, our solution breaks the virtualization boundary and captures game frames in place. Figure 10 plots the workflow of GFI. The high-level idea is to leverage cross-layer information from both the guest system and the host-side virtual GPU to intercept and forward *handles* (as opposed to the actual frame content), so as to minimize data copies. In detail, we customize the `gralloc` graphics allocator [12], the guest GPU driver, along with the virtual GPU device in Trinity. `Gralloc` provides crucial information regarding frame correlations across the virtualization boundary, so that we can identify the actual host-side GPU resources using the guest-side frame identifier. Meanwhile, in order to separate render instructions of the target game from other apps like Android system UI, we modify the guest GPU driver, adding Android framework-level `Surface` information into GPU render contexts.

Based on the above cross-layer information, we monitor

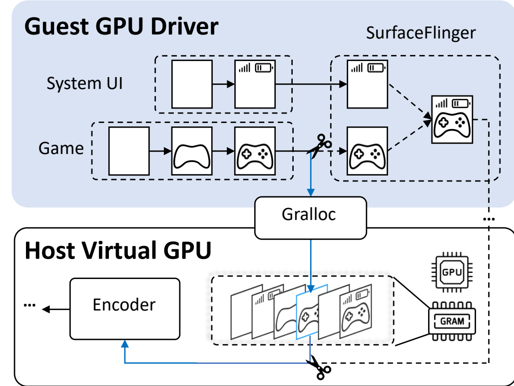


Figure 10: The workflow of GFI.

frame swap events (`eglSwapbuffers`) in the GPU driver, so that when a game-owned `Surface` delivers a frame to `SurfaceFlinger`, we can extract the actual resource handle(s) referring to the game frame in the host-side virtual GPU device. Afterwards, the intercepted resource handle(s) are immediately forwarded to the video encoder module, which utilizes the *interop* libraries provided by the GPU vendor (e.g., `nvenc` [60] for NVIDIA GPUs) to achieve in-place colorspace conversion and video encoding. In this way, data exchange between the CPU and the GPU, as well as between the guest and the host, is kept minimal.

With the game frames intercepted, Stages 6-9 (c.f. §3.2) in the original graphics pipeline can be bypassed, including two VSync events, Layer Composition I, and Virtual Display. Additionally, for `SurfaceFlinger` and the virtual display device in Trinity, we disable their rendering logic to reduce the GPU load, since the composited frames are no longer needed (frame swapping is preserved to not disrupt the data flow). Moreover, to support a minority of games that use multiple `Surfaces` for video playback or UI overlay, we add a trimmed-down compositor in the encoder to replace the original Android compositor in the VM, since the original Android compositor introduces considerable overheads (e.g., computing visible regions, matching display devices, and drawing layers). Specifically, we customize the Android compositor by 1) discarding game-irrelevant layers, 2) removing unnecessary post-processing (e.g., submitting frames to display), and 3) synchronizing the composition pace with the encoder to minimize latency overheads. Instead of relying on a standalone VSync (which defeats our original purpose of bypassing VSync events), frames are cached and composited right before encoding, so the delay caused by frame composition in these corner cases can be essentially reduced.

4.3 Remote VSync Coordinator (RVC)

Design Overview. With the informative help of GFI (Game Frame Interceptor), RVC aims to align the remaining three VSync events (VSync1, VSync4, and VSync5). As VSync5 is controlled by the client OS and cannot be changed by a user

app, RVC proactively synchronizes the timing of the cloud-side VSync1 and VSync4 with the client-side VSync5. To this end, RVC first obtains the timing information of VSync5 from the Android frame pacing library [6] in the client[3], and then predicts the delay of each intervening stage between VSync1 and VSync5 based on hierarchical time series forecasting [85] **(Hierarchical Latency Prediction)**.

Afterwards, RVC performs Synergetic VSync Alignment which exploits the latency forecasts to adaptively postpone VSync1 **(Synergetic VSync Alignment for VSync1)** and strategically perform reactive frame encoding by decoupling VSync4 **(Synergetic VSync Alignment for VSync4)**. Hence, a game frame (whose rendering and encoding are driven by VSync1 and VSync4 respectively) can arrive at the client (more specifically, arrive at VSync5) just in time, without waiting for the signaling of VSync4 and VSync5.

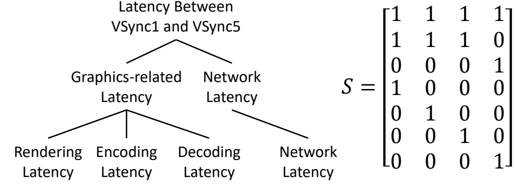
Hierarchical Latency Prediction. There are four stages happening between VSync1 and VSync5: game rendering, frame encoding, network transmission, and frame decoding. Although the latency of each stage can be independently forecast as a time series and summed directly to get the overall latency, the prediction result is oftentimes highly biased due to error propagation [39].

To minimize the prediction error, our key insight is that the stages have latent correlations within the end-to-end graphics pipeline. For example, if the game player suddenly enters a complex scene, game rendering will take longer time, the encoder will need more time to encode the frame as a standalone key frame (i.e., I frame) [74], and the encoded frame will be larger in size, leading to longer network transmission time and longer client-side frame decoding time. Also, we note that the involved latencies can be hierarchically organized as shown in Figure 11. Therefore, we adopt hierarchical time series forecasting [85] based on the MinT approach [85].

First, we adopt regression trees [52] to independently forecast the time series at all levels of the hierarchy (termed *base forecasts*). We resort to the regression tree algorithm as opposed to other algorithms because it is effective for both series with seasonal features [67] and fluctuating series like network latencies [89]. Moreover, it incurs low computation (<0.5 *ms*) and memory (<1 *MB*) overhead. Efficiency is of particular significance in our case, since the predictions need to be done at intervals of a few seconds.

Regarding the inputs of the regression trees, we follow common practice [67, 89] and feed them with the most recent latency data, as well as relevant data including the size of the encoded frame, network loss rate, and network bandwidth passively estimated from the media stream [89]. Since we need to make predictions at the frame-level granularity, the

[3]The timings of client-side stages (e.g., packet receiving, decoding, and VSync5) are synchronized with the cloud server's clock, leveraging a widely-used clock offset estimation approach [22] and a classic skew compensation algorithm [56]. The historical latency series (including the one-way network latency) are measured according to the calibrated timestamps.



$$S = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 11: The tree and the matrix representation of the hierarchical latency series.

time window (i.e., the time unit in the forecast) is set as the smallest VSync interval among VSync1, VSync4 and VSync5 (typically 1/60 *s*). Accordingly, the information window size (i.e., the number of history time windows) and the forecast horizon (i.e., the number of future time windows) are typically set as 240 (=60×4) and 60, respectively. More details about the setting of hyperparameters will be provided later in §5.4.

Given the base forecasts, we aim at minimizing the forecast errors by strategically reconciling the forecasts across the levels of the latency hierarchy. Let $\mathcal{D}_t = \{y_1, y_2, ..., y_t\}$ denote the history data observed up to time t, where y_i $(i = 1, 2, ..., t)$ is a vector containing all latencies in all levels of the hierarchy at time i. Let $\hat{y}_t(k)$ $(k = 1, 2, ..., K)$ denote forecasts of k future time windows based on $\mathcal{D}_t$, where k is the forecast horizon. Then, the optimal forecasts of k future time windows minimizing the conditional expectation

$$E[\| \, y_{t+k} - \hat{y}_t(k) \, \|_2^2 | \mathcal{D}_t], \tag{1}$$

where $\| \cdot \|_p$ is an L_p norm [30], is derived from

$$\mu_t(k) = E[y_{t+k} | \mathcal{D}_t]. \tag{2}$$

More details on optimal hierarchical forecasts are in [39].

If the conditional expectation (1) has errors, we wish to provide the best estimate of $\mu_t(k)$. Specifically, from the base forecasts $\hat{y}_t(k)$, we compute the revised forecasts as

$$\tilde{y}_t(k) = SP\hat{y}_t(k) \tag{3}$$

with a weight matrix $P \in \mathcal{P} \subseteq \mathbb{R}^{4\times7}$, where $\mathcal{P}$ is the domain of P and S is a "summing" matrix of order 7×4 used to aggregate the lowest-level latency series, as shown in Figure 11. In detail, we compute the weight matrix P by solving the following optimization problem:

$$\min_{P \in \mathcal{P}} Tr(Var[y_{t+k} - \tilde{y}_t(k) | \mathcal{D}_t]) \text{ subject to } SPS = S; \tag{4}$$

$Tr(\cdot)$ is the trace of a matrix, based on MinT estimator [85].

To conclude, we realize combination forecasts for all levels of the latency series hierarchy through an optimal weight matrix P. In this way, we minimize the standard error of the coherent forecasts across the entire hierarchy.

Synergetic VSync Alignment for VSync1. With the predicted latencies in hand, the paces of VSync events can be adjusted in a synergetic fashion. In practice, we adjust the timing of VSync1 by blocking game rendering in the virtual GPU. As demonstrated in Figure 12, properly adjusting VSync1 can

reduce the interactive latency, by making the rendered frames reflect more recent user inputs (e.g., the first three arrows in Input Series 2). Specifically, once alignment occurs, user inputs that happen in the interval highlighted by the red box (i.e., the first 3 inputs in Input Series 2) will be processed by "Rendering1" instead of by the original "Rendering2". Consequently, the response of these user inputs can be displayed earlier, leading to a lower interactive latency.

We use Monte Carlo sampling [71] to minimize the delay expectation of n future frames in the pipeline, where n equals the forecast horizon. Specifically, we let V_c denote the timestamp of the most recent VSync5 event in the client, and let $x = \{x_1, x_2, ..., x_n\}$ denote predicted timestamps when the client finishes frame decoding, where x_i is the timestamp of the i-th future frame from now on. Then, we denote the timestamps of subsequent relevant VSync5 events as $V = \{V_1, V_2, ..., V_n\}$, where V_i is computed as

$$V_i = V_c + \left(1 + \left\lfloor \frac{x_i - V_c}{I_c} \right\rfloor\right) \cdot I_c, i = 1, 2, ..., n. \quad (5)$$

Here I_c is the time interval of VSync5. Given x and V, the problem is to obtain the minimum positive value of o (denoting the offset of the current blocking adjustment) that minimizes $E(x,V,o)$, which is the latency expectation of the subsequent n frame processings. It is computed as

$$E(x,V,o) = \sum_{i=1}^{n} f(x_i, V_i, o). \quad (6)$$

Here $f(x_i, V_i, o)$ is the expected delay induced by the VSync5 event on V_i, which is computed as

$$f(x_i, V_i, o) = (I_c - (V_i - x_i - o)) \mod I_c. \quad (7)$$

Given that the latency expectation (6) is a discontinuous function, we approximate the optimal value of o based on Monte Carlo sampling. Finally, the VSync1 events are postponed with an offset of o, completing the alignment. In reality, we perform the alignment for VSync1 periodically every n frames, where n is the forecast horizon.

Synergetic VSync Alignment for VSync4. Given that the cloud-side game frame rate and the client-side video frame rate might be different, frame encoding is normally driven by VSync4, whose interval matches the video frame rate. However, with the help of GFI, game frames can be encoded in a reactive manner without waiting for VSync4, while maintaining client-side frame rate stability based on the forecasts.

RVC decouples frame encoding from VSync4 events to reactively encode ready-to-consume frames in time. When the cloud-side game frame rate is lower than or equal to the client-side video frame rate, all produced game frames need to be encoded for streaming to satisfy the client-side requirement as much as possible. Otherwise, the forecasts are used to determine whether the encoder should encode or drop a game frame, to match the video frame rate requested by the user.

The encoder memorizes the timeline of the original VSync4 events, as well as the timestamp of the last encoded frame.

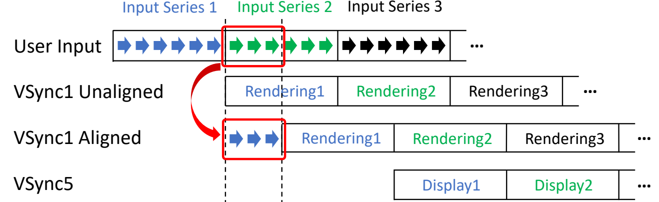


Figure 12: An example of Synergetic VSync Alignment. After the alignment, the first three inputs in Input Series 2 are processed ahead of schedule.

When a new game frame F_i arrives at T_i, it is encoded if the age of the last encoded frame exceeds one VSync interval, or if the predicted arrival time T_{i+1} of the next frame F_{i+1} satisfies $|T_{i+1} - T_v| \geq |T_i - T_v|$ (T_v is when the next VSync4 event occurs). If neither condition holds, F_i will be dropped since F_{i+1} better fits the client-side video pace. In this way, we can avoid delays incurred by VSync4 while adaptively maintaining the frame rate stability of the video.

5 Evaluation

We first describe our evaluation setup in §5.1, followed by evaluating the overall performance of LoopTailor in §5.2 and assessing the contributions of individual modules in §5.3. In §5.4, we present several micro-benchmarks to quantify the impact of key parameters and network dynamics.

5.1 Experimental Setup

Collaborating with X-MCG, we deploy LoopTailor on their cloud infrastructure. This facilitates fair comparisons between the original X-MCG and X-MCG enhanced with LoopTailor, which we refer to as LoopTailor for brevity. We also compare LoopTailor with other representative MCG platforms studied in §2.1: Tencent Pioneer, NetEase Cloud Gaming, JoyArk, and CloudMoon. Regarding client devices, we leverage the 100 mobile devices used in our measurement study (§2.1). Other settings (including camera settings, game types, game play duration, RAT, streaming resolution / frame rate, etc.) are the same as those in §2.1. We use the 100 mobile devices to collect evaluation data during a whole month (Nov. 2023), and collected a total of 21,743 valid records.

We also implement two intuitive baseline solutions based on the X-MCG architecture to evaluate the performance of LoopTailor in more depth. (1) Disable VSync: Since the client-side VSync5 and the in-game VSync1 cannot be controlled by an MCG platform, we disable VSync2/3/4 in the cloud. (2) In-VM Streaming: With the help of virtualized codec and NIC devices, videos are streamed in the guest OS (though the host NIC should still be used).

5.2 Overall Performance

Figure 13 plots the average interactive latency for X-MCG, LoopTailor, and four other MCG platforms (refer to Table 1 for their name-ID mappings), with a breakdown between

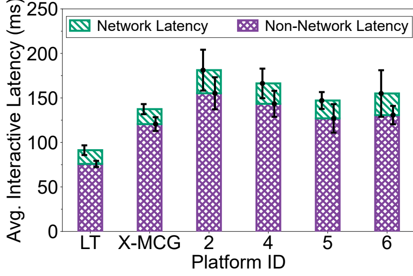


Figure 13: Average interactive latency. LT denotes LoopTailor.

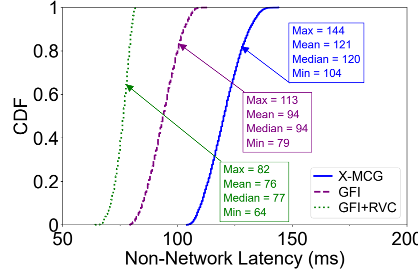


Figure 14: Performance breakdown with regard to non-network latency.

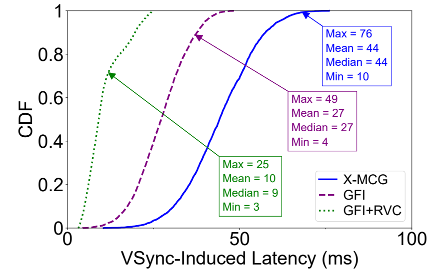


Figure 15: Performance breakdown regarding VSync-induced latency.

Table 4: Latency comparison (in milliseconds) between LoopTailor and baseline solutions. "Tail" refers to 99% percentile.

Solutions	Interactive Latency			Non-Network Latency		
	Avg.	Med.	Tail	Avg.	Med.	Tail
LoopTailor	91	91	95	76	77	82
Disable VSync	106	105	121	90	90	104
In-VM Streaming	141	139	162	125	124	146

the network vs. non-network latency. As shown, LoopTailor yields the lowest average interactive latency of 82-96 *ms* (91 *ms* on average, 95 *ms* in 99% percentile), 35% lower than the original X-MCG (139 *ms* on average, 158 *ms* in 99% percentile). Recall that a smooth gaming experience requires a sub-100 *ms* interactive latency, which is achieved by LoopTailor. Note that the desirable interactive latency of LoopTailor is also ascribed to X-MCG's wide deployment of edge servers in the latter half of 2023. The non-network latency of LoopTailor is only 64-82 *ms* (76 *ms* on average, 82 *ms* in 99% percentile), 37% lower than the original X-MCG (121 *ms* on average, 141 *ms* in 99% percentile) and 40% lower than the best of the other four MCG platforms (127 *ms* on average, 150 *ms* in 99% percentile).

As shown in Table 4, LoopTailor outperforms the baseline solutions in terms of interactive latency and non-network latency. Although it appears that LoopTailor has no significant latency advantage over Disable VSync, the latter introduces severe side effects including unstable frame rate and screen tearing [42]. As for In-VM Streaming, it performs the worst due to the additional overheads of virtualized codec and NIC devices for multiplexing hardware resources.

Regarding other QoE metrics for cloud gaming, we observe that the average client-side video frame rate stays high at an average of 59.8 FPS, with a small standard deviation of 0.6. This indicates that LoopTailor is capable of delivering a smooth game streaming experience. Regarding the image quality of LoopTailor, we do not observe jittery frames or degradation of image quality compared to original X-MCG.

5.3 Contributions of Individual Modules

We evaluate the performance gain brought by each module of LoopTailor: GFI (Game Frame Interceptor) and RVC (Remote VSync Coordinator), by enabling them incrementally. We exclude the network latency when reporting the results.

We begin with only enabling GFI. Recall that GFI eliminates VSync2 and VSync3. As shown in Figure 14, this results in an average non-network latency of 94 *ms*, a 22% reduction compared to X-MCG. The *maximum* non-network latency of GFI-only LoopTailor is even 7% lower than the *average* non-network latency of X-MCG. To delve deeper into benefits of GFI, we measure the latency reduction of different stages. First, we consider the latency incurred by only VSync events (termed VSync-induced latency) and plot its distributions in Figure 15. As shown, GFI reduces the VSync-induced latency by 38% on average. Second, GFI helps cut the layer composition latency by 52% since several unnecessary pipeline stages are bypassed. Third, GFI further helps accelerate game rendering by 6% because with a few pipeline stages bypassed, more resources can be allocated for rendering.

Next, we enable RVC to make a full-fledged LoopTailor system. As in Figure 14, RVC further reduces the average non-network latency by 19%, compared to using GFI alone. This translates to a 37% reduction in the total non-network latency compared to X-MCG. Through Synergetic VSync Alignment, RVC greatly reduces the average VSync-induced latency by 57% (73%) compared to using GFI alone (X-MCG). Unlike GFI which only optimizes the cloud side, RVC tackles both the cloud and the client side. It reduces the average latency incurred by VSync5 on the client side from 8 *ms* to 3 *ms*.

The above results demonstrate both GFI and RVC are essential for LoopTailor. Their synergy allows LoopTailor to reach a consistently low non-network latency of 64-82 *ms*.

5.4 Micro-benchmarks

Impact of the Information Window Size. Recall from §4.3 that the information window size (IWS) determines the number of time windows used in latency prediction. Figure 16 shows the relationship between the IWS and the standard error of latency prediction (the forecast horizon is set to 1 in this experiment). As shown, the standard error first decreases and then increases. The decrease is because larger information windows usually provide more sufficient observations on history data, whereas the increase is explained by too many outdated history data within the window.

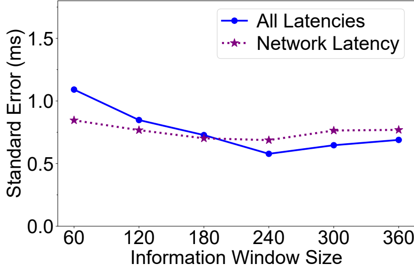


Figure 16: Avg. std error of RVC with different information windows.

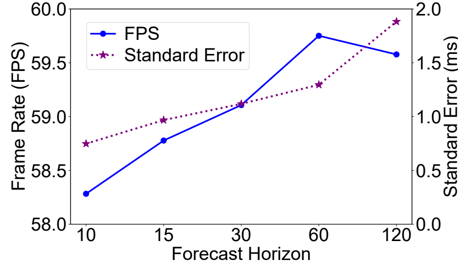


Figure 17: Avg. video frame rate & std error with different forecast horizons.

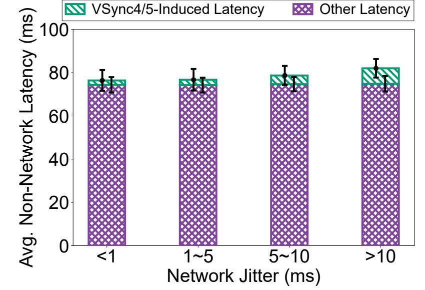


Figure 18: Avg. non-network latency with different network jitters.

Given the existence of key frames in game rendering, it is preferable to set the IWS to be larger than the intervals between each pair of key frames (usually about 10 to 20 VSync intervals based on our measurements). Meanwhile, IWS should not be too large considering both the forecasting errors and computation overheads. Figure 16 shows that RVC achieves the lowest standard error when IWS is around 240, which reaches the minimum standard error (0.6 *ms*) with a low computation overhead (<0.5 *ms*). Regarding the prediction of network latency, RVC achieves an average standard error of <1 *ms* (0.69-0.84 *ms* as shown in Figure 16) under different IWS settings. The standard errors exhibit tiny fluctuations given that the network condition in our evaluation is stable with only ∼2 *ms* jitters. The performance of LoopTailor under diverse network conditions is discussed later in this section.

Impact of the forecast horizon. The forecast horizon determines the number of future time windows produced by a prediction in RVC. On one hand, a smaller forecast horizon leads to a higher frequency of VSync1 adjustments, as illustrated in §4.3; adjusting VSync1 too frequently may result in a slight client-side video frame rate drop, because an incorrect forecast may cause a frame deadline miss at the client. On the other hand, a large forecast horizon may increase the forecasting error. We experimentally quantify this tradeoff in Figure 17, where the IWS is set to 240. Note that the slight drop in the frame rate at a forecast horizon of 120 is due to the sharp increase in the standard error. Based on the results, we select a forecast horizon of 60 to balance the above tradeoff.

Impact of Network Jitters. LoopTailor should ideally maintain a low non-network latency even when the network condition is highly dynamic. Figure 18 shows the average non-network latency under different network jitters (i.e., standard deviations of network latencies). The results demonstrate the robustness of LoopTailor under diverse network conditions: with ≤10 *ms* network jitters, there is only negligible fluctuation in the non-network latency (75-78 *ms*); even when the network jitter exceeds 10 *ms*, the non-network latency only increases marginally to 82 *ms*. The slight increase in the non-network latency is caused by the inflation of VSync4/5-induced latency, which is attributed to the elevated forecasting error under highly fluctuating network conditions.

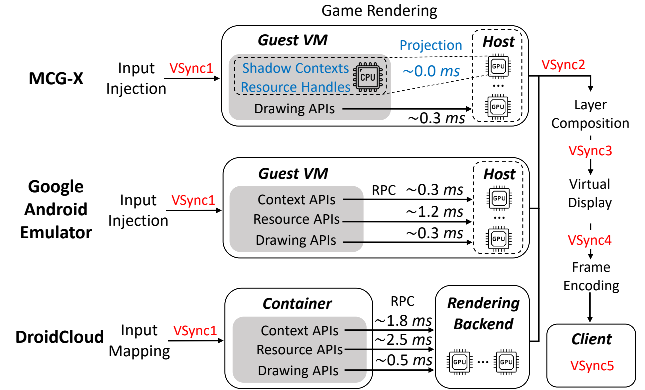


Figure 19: Architectural comparison of X-MCG and other representative MCG solutions. The numbers atop arrows refer to the average latency overhead per game frame.

6 Discussion

Generalizability. VSync1∼5 are not specific to X-MCG; in fact, it is impractical to disable them as they are necessitated by the design of the Android system (on both cloud and client sides) and the nature of constant-frame-rate video encoding (see §3.2). Thus, the defects of X-MCG are common to every Android- and video streaming-based MCG platform. Figure 19 illustrates a detailed comparison of the architectures of X-MCG and other representative MCG solutions. As shown, employing other solutions such as Google Android Emulator-based (abbreviated as GAE-based) and DroidCloud [47] also results in the five VSync events in the end-to-end graphics pipeline. Given the above facts, we believe that our insights and designs are widely applicable to other platforms.

The developers of X-MCG make a careful choice among existing MCG solutions by conducting comprehensive benchmarks on them. The experiment results reveal that the graphics virtualization mechanism of Trinity introduces ∼0.3 *ms* latency overhead per frame, 83%-94% less than other representative solutions [33]. Additionally, Sunshine exhibits 12.6%-26.7% lower end-to-end latency compared to other streaming tools [37, 59, 63] due to its efficient in-place frame processing capabilities. However, these advanced virtualization and frame processing techniques still need the VSync

mechanism for frame access synchronization with other stages in the graphics pipeline. Worse still, the high efficiency of rendering and frame processing of X-MCG may even amplify the impact of VSync events on the interactive latency.

Although we were unable to scrutinize other mobile OSes such as iOS, we believe our insights and design principles still apply, given that VSync is a graphics primitive and that the virtualization stack is widely used for the cloud gaming backend. In fact, the closed-source nature of iOS prompts many MCG applications running on iOS to leverage virtualized Android backends to stream mobile games [41]. Furthermore, iOS also uses layer compositors (e.g., Core Animation), and iOS cloud gaming backends must also use virtualization to achieve resource multiplexing.

Application scope. MCG platforms generally focus on enabling users with middle/low-end devices to smoothly play graphics-demanding mobile games. For budget reasons, the refresh rates of such devices are typically fixed and limited, allowing LoopTailor to fully exploit its capability to optimize the interactive latency for mainstream MCG users. For high-end user devices with adaptive/high refresh rates, things are different for VSync5 as the intervals between adjacent VSync5 events become variable/short.[4] In such cases, only part of RVC (i.e., Synergetic VSync Alignment for VSync1) is less effective. Other LoopTailor components are still effective, as VSync1∼4 are not on client devices.

7 Related Work

Cloud Gaming Systems. Based on where rendering instructions are executed, cloud gaming systems can be classified into two categories: host-rending and client-rendering. Host-rendering systems typically follow the thin-client architecture [19]. It offloads computation-intensive tasks to cloud servers, and livestreams rendered game frames as an encoded video to thin clients for display. Many systems follow this paradigm such as those proposed by the academia [37, 40, 47, 48, 64, 76, 86, 88] and the industry [4, 21, 55, 58, 61, 75, 78, 79].

Client-rendering systems, such as Games@Large [57] and LiveRender [49], intercept graphics instructions in the cloud and send them (along with graphics data such as vertices and textures) to the client for execution. While this paradigm can deliver high-fidelity graphics and enable secure GPU computing [38], it requires powerful client devices and networks [77], making it less attractive to low-end devices. Our work focuses on optimizing the performance of host-rendering systems given their dominant popularity.

Android in the Cloud. Landing Android onto cloud systems has become an emerging research topic. Existing cloud solutions typically run Android in virtual machines [7, 8, 91] or containers [21, 47, 76]. Considering the generality of both the Game Rendering Interceptor and Remote VSync Coordinator, we believe LoopTailor applies to most of the above systems. These systems generally use API remoting [27] or device emulation [65, 69] for graphics virtualization, which can be laggy and inefficient due to, for example, excessive guest-host interactions [33]. To overcome such limitations, recent advances such as Trinity [33] (upon which we built LoopTailor) and vSoC [66] employ various techniques to reduce guest-host interactions, leading to higher graphics performance.

Another body of works set out to adapt Android to cloud environments. For instance, DroidCloud [47] enables Android to render on off-system GPU pools; CARE [76] cloudifies Android into a cloud-native system by streamlining Android system services. Our work instead operates on the graphics pipeline, and is orthogonal to the above research.

Latency Mitigation for Real-time Communication. Beyond VSync, there is a plethora of works on reducing the latency (incurred by rendering, network, and video coding) for real-time communication (RTC) systems such as MCG. They investigate a wide range of latency optimization dimensions including rendering acceleration [25, 68], network congestion control [5, 84], video coding [36, 73, 88], optimization model/metric [35, 93, 94], and cloud architecture [46, 50, 70]. Regarding graphics-related latency, recent studies [45, 86] attempt to reduce it by predicting video frames, but they may fall short in real-world conditions due to the high randomness of user actions [45, 86]. Contrasting above efforts, our work focuses on shortening the lengthy cloud-side graphics pipeline through a new dimension of regulating VSync events.

8 Conclusion

Mobile cloud gaming, as a highly interactive immersive application, faces new latency challenges due to its sophisticated backend graphics processing. In this paper, we identify VSync as a major contributor to the end-to-end interactive latency overhead and show that a specialized design can effectively minimize the VSync overhead. The result is a stable sub-100 *ms* interactive latency, leading to an average reduction of 35% compared with those provided by state-of-the-art MCG platforms. We believe that the ideas and key building blocks of LoopTailor, such as in-place frame capture and multi-stage latency prediction, have more profound implications to the design and implementation of emerging immersive applications, especially those operating in the edge/cloud-based model.

Acknowledgments

We are grateful to our shepherd, Zili Meng, for his invaluable feedback. We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703, the National Natural Science Foundation of China under grants 62332012 and 62472245, and the Ant Group.

[4] In Android, variable refresh rate is supported by switching between multiple discrete refresh rates rather than continuously adapting refresh rates (as of Sep. 2024) [1], and thus the client-side VSync5 always exists.

References

[1] Ady Abraham. High Refresh Rate Rendering on Android, 2020. https://android-developers.googleblog.com/2020/04/high-refresh-rate-rendering-on-android.html.

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. CRC Press, August 2018.

[3] Ahmad Alhilal, Tristan Braud, Bo Han, and Pan Hui. Nebula: Reliable Low-Latency Video Transmission for Mobile Cloud Gaming. In *Proc. of ACM WWW*, pages 3407–3417, 2022.

[4] Amazon. Luna Cloud Gaming Service, 2024. https://amazon.com/luna/landing-page?ref=ln_social.

[5] Maryam Amiri, Hussein Al Osman, Shervin Shirmohammadi, and Maha Abdallah. An SDN Controller for Delay and Jitter Reduction in Cloud Gaming. In *Proc. of ACM MM*, pages 1043–1046, 2015.

[6] Android Developers. Android Frame Pacing Library, 2023. https://developer.android.com/games/sdk/frame-pacing.

[7] Android Developers. Cuttlefish Virtual Android Devices, 2023. https://source.android.com/docs/setup/create/cuttlefish.

[8] Android Developers. Google Android Emulator, 2023. https://developer.android.com/studio/run/emulator.

[9] Android Developers. Surface API, 2023. https://developer.android.com/reference/android/view/Surface.

[10] Android Developers. SurfaceFlinger and WindowManager, 2023. https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager.

[11] Android Developers. Android Open Source Project, 2024. https://source.android.com/.

[12] Android Developers. BufferQueue and Gralloc, 2024. https://source.android.com/docs/core/graphics/arch-bq-gralloc.

[13] Android Developers. Hardware Composer HAL, 2024. https://source.android.com/docs/core/graphics/implement-hwc.

[14] Android Developers. Layers and Displays, 2024. https://source.android.com/docs/core/graphics/layers-displays.

[15] Android Developers. Media Projection API, 2024. https://developer.android.com/guide/topics/large-screens/media-projection.

[16] Android Developers. OpenGL ES Overview, 2024. https://developer.android.com/develop/ui/views/graphics/opengl/about-opengl.

[17] Android Developers. Screen Capturing and Sharing, 2024. https://developer.android.com/about/versions/lollipop/android-5.0.

[18] Android Developers. The VSync Mechanism, 2024. https://source.android.com/docs/core/graphics/implement-vsync.

[19] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proc. of ACM SOSP*, pages 277–290, 2005.

[20] Boosteroid. Boosteroid Cloud Gaming, 2024. https://boosteroid.com/.

[21] Canonical. Anbox Cloud - Scalable Android in the Cloud, 2024. https://anbox-cloud.io.

[22] Qasim M. Chaudhari, Erchin Serpedin, and Khalid Qaraqe. On Maximum Likelihood Estimation of Clock Offset and Skew in Networks With Exponential Delays. *IEEE Transactions on Signal Processing*, 56:1685–1697, 2008.

[23] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-Gaming: A Cost-Efficient Cloud Gaming System at Scale. *IEEE Transactions on Parallel and Distributed Systems*, 30:2849–2865, 2019.

[24] China Mobile and ZTE. Powered by Sa: 5g Mec-Based Cloud Game Innovation Practice. Technical report, 2020.

[25] Shen Ciao, Zhongyue Guan, Qianxi Liu, Li-Yi Wei, and Zeyu Wang. Ciallo: GPU-Accelerated Rendering of Vector Brush Strokes. In *Proc. of ACM SIGGRAPH*, pages 1–11, 2024.

[26] CloudMoon. CloudMoon - Cloud Gaming, 2024. https://play.google.com/store/apps/details?id=com.nianwei.cloudphone&hl=en_US.

[27] Micah Dowty and Jeremy Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43:73–82, 2009.

[28] Epic Games. Frame Pacing for Mobile Devices, 2024. https://docs.unrealengine.com/5.3/en-US/frame-pacing-for-mobile-devices-in-unreal-engine/.

[29] Epic Games. Unreal Engine Official Website, 2024. https://www.unrealengine.com/en-US.

[30] Richard Farebrother. *L1-Norm and L∞-Norm Estimation: An Introduction to the Least Absolute Residuals, the Minimax Absolute Residual and Related Fitting Procedures*. Springer Science & Business Media, April 2013.

[31] Ilja T Feldstein and Stephen R Ellis. A Simple Video-based Technique for Measuring Latency in Virtual Reality or Teleoperation. *IEEE Transactions on Visualization and Computer Graphics*, 27(9):3611–3625, 2020.

[32] James D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1996.

[33] 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 *Proc. of USENIX OSDI*, pages 285–301, 2022.

[34] Sumanta Guha. *Computer Graphics Through OpenGL®: From Theory to Experiments*. CRC Press, December 2018.

[35] Pouya Hamadanian, Doug Gallatin, Mohammad Alizadeh, and Krishna Chintalapudi. Ekho: Synchronizing Cloud Gaming Media across Multiple Endpoints. In *Proc. of ACM SIGCOMM*, pages 533–549, 2023.

[36] Luke Hsiao, Brooke Krajancich, Philip Levis, Gordon Wetzstein, and Keith Winstein. Towards Retina-quality VR Video Streaming: 15ms Could Save You 80% of Your Bandwidth. *ACM SIGCOMM Computer Communication Review*, 52(1):10–19, 2022.

[37] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of ACM MMSys*, pages 36–47, 2013.

[38] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *Proc. of USENIX NSDI*, pages 817–833, 2020.

[39] Rob J. Hyndman, Roman A. Ahmed, George Athanasopoulos, and Han Lin Shang. Optimal Combination Forecasts for Hierarchical Time Series. *Computational Statistics & Data Analysis*, 55:2579–2589, 2011.

[40] Iryanto Jaya, Yusen Li, and Wentong Cai. Improving Scalability, Sustainability and Availability via Workload Distribution in Edge-Cloud Gaming. In *Proc. of ACM MM*, pages 2987–2995, 2022.

[41] JoyArk. JoyArk – Explore and Play Games Instantly, 2024. https://joyark.com/.

[42] Viktor Kelkkanen, Markus Fiedler, and David Lindero. Synchronous Remote Rendering for VR. *International Journal of Computer Games Technology*, 2021(1):6676644, 2021.

[43] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment Anything. *arXiv:2304.02643*, 2023.

[44] David Large and James Farmer. *Modern Cable Television Technology*. Elsevier, 2004.

[45] 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 *Proc. of ACM MobiSys*, pages 151–165, 2015.

[46] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, Chen Sun, Gareth Tyson, and Hongqiang Harry Liu. LiveNet: A Low-Latency Video Transport Network for Large-Scale Live Streaming. In *Proc. of ACM SIGCOMM*, pages 812–825, 2022.

[47] Linsheng Li, Bin Yang, Cathy Bao, Shuo Liu, Randy Xu, Yong Yao, Mohammad R. Haghighat, Jerry W. Hu, Shoumeng Yan, and Zhengwei Qi. DroidCloud: Scalable High Density AndroidTM Cloud Rendering. In *Proc. of ACM MM*, pages 3348–3356, 2020.

[48] Yusen Li, Haoyuan Liu, Xiwei Wang, Lingjun Pu, Trent Marbach, Shanjiang Tang, Gang Wang, and Xiaoguang Liu. Themis: Efficient and Adaptive Resource Partitioning for Reducing Response Delay in Cloud Gaming. In *Proc. of ACM MM*, pages 491–499, 2019.

[49] Li Lin, Xiaofei Liao, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. In *Proc. of ACM MM*, pages 347–356, 2014.

[50] Yuhua Lin and Haiying Shen. CloudFog: Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service. *IEEE Transactions on Parallel and Distributed Systems*, 28:431–445, 2017.

[51] LizardByte. Sunshine Official Website, 2023. https://docs.lizardbyte.dev/projects/sunshine/en/latest/about/overview.html.

[52] Wei-Yin Loh. Classification and Regression Trees. *WIREs Data Mining and Knowledge Discovery*, 1:14–23, 2011.

[53] Sanjay Madhav. *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach*. Pearson Education, 2014.

[54] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling High Quality Real-Time Communications with Adaptive Frame-Rate. In *Proc. of USENIX NSDI*, 2023.

[55] Microsoft. Xbox Cloud Gaming, 2024. https://www.xbox.com/en-US/cloud-gaming.

[56] S.B. Moon, P. Skelly, and D. Towsley. Estimation and Removal of Clock Skew from Network Delay Measurements. In *Proc. of IEEE INFOCOM*, pages 227–234 vol.1, 1999.

[57] Itay Nave, Haggai David, Alex Shani, Yoav Tzruya, Arto Laikari, Peter Eisert, and Philipp Fechteler. Games@large Graphics Streaming Architecture. In *Proc. of IEEE ISCE*, pages 1–4, 2008.

[58] Netease. Netease Cloud Gaming, 2024. https://cg.163.com/.

[59] NVIDIA. NVIDIA GameStream, 2022. https://www.nvidia.com/en-us/support/gamestream/.

[60] NVIDIA. NVENC Video Codec SDK, 2024. https://developer.nvidia.com/video-codec-sdk.

[61] NVIDIA. NVIDIA GeForce NOW, 2024. https://www.nvidia.com/en-us/geforce-now/.

[62] NVIDIA DOCS HUB. Single Root IO Virtualization (SR-IOV), 2024. https://docs.nvidia.com/doca/sdk/single+root+io+virtualization+(sr-iov)/index.html.

[63] Open-Stream Developers. Open-Stream Game Streaming Platform, 2024. https://open-stream.net/.

[64] Mikko Pitkänen, Marko Viitanen, Alexandre Mercat, and Jarno Vanne. Remote VR Gaming on Mobile Devices. In *Proc. of ACM MM*, pages 2191–2193, 2019.

[65] Qemu Developers. Documentation of Virtio-Gpu, 2023. https://www.qemu.org/docs/master/system/devices/virtio-gpu.html.

[66] Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, and Tianyin Xu. vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware. In *Proc. of ACM SOSP*, 2024.

[67] Sai Swaroop Ratakonda and Sreela Sasi. Seasonal Trend Analysis on Multi-Variate Time Series Data. In *Proc of IEEE ICDSE*, pages 1–6, 2018.

[68] Eric Risser. Rendering 3D Volumes Using Per-pixel Displacement Mapping. In *Proc. of ACM SIGGRAPH Symposium on Video Games*, pages 81–87, 2007.

[69] Rusty Russell. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42:95–103, 2008.

[70] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, and Bruce Maggs. DChannel: Accelerating Mobile Applications With Parallel High-Bandwidth and Low-Latency Channels. In *Proc. of USENIX NSDI*, pages 419–436, 2023.

[71] Alexander Shapiro. Monte Carlo Sampling Methods. In *Handbooks in Operations Research and Management Science*, volume 10 of *Stochastic Programming*, pages 353–425. Elsevier, January 2003.

[72] Ryan Shea, Di Fu, and Jiangchuan Liu. Cloud Gaming: Understanding the Support From Advanced Virtualization and Hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 25:2026–2037, 2015.

[73] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. Using Graphics Rendering Contexts to Enhance the Real-Time Video Coding for Mobile Cloud Gaming. In *Proc. of ACM MM*, pages 103–112, 2011.

[74] W. C. Siu. *Multimedia Information Retrieval and Management: Technological Fundamentals and Applications*. Springer Science & Business Media, January 2003.

[75] Sony. PlayStation Plus Cloud Gaming, 2024. https://www.playstation.com/en-us/games/playstation-plus-essential-1-month-subscription.

[76] Dongjie Tang, Cathy Bao, Yong Yao, Chao Xie, Qiming Shi, Marc Mao, Randy Xu, Linsheng Li, Mohammad R. Haghighat, Zhengwei Qi, and Haibing Guan. CARE: Cloudified Android OSes on the Cloud Rendering. In *Proc. of ACM MM*, pages 4582–4590, 2021.

[77] Dongjie Tang, Yun Wang, Linsheng Li, Jiacheng Ma, Xue Liu, Zhengwei Qi, and Haibing Guan. gRemote: API-Forwarding Powered Cloud Rendering. In *Proc. of ACM HPDC*, pages 197–201, 2020.

[78] Tencent. Tencent Pioneer, 2024. https://gamer.qq.com/.

[79] Tencent. Tencent Start, 2024. https://start.qq.com.

[80] Tencent Cloud. Tencent Cloud Gaming Product Documentation, 2023. https://cloud.tencent.com/document/product/1162/46097.

[81] Unity Technologies. Unity Frame Pacing for Android, 2024. https://docs.unity3d.com/ScriptReference/Application-targetFrameRate.html.

[82] Unity Technologies. Unity Official Website, 2024. https://unity.com.

[83] Alexander Van't Hof, Hani Jamjoom, Jason Nieh, and Dan Williams. Flux: Multi-Surface Computing in Android. In *Proc. of ACM EuroSys*, pages 1–17, 2015.

[84] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, Changxi Zheng, Jing Wang, and Honghao Liu. Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. In *Proc. of USENIX NSDI*, pages 113–129, 2024.

[85] Shanika L. Wickramasuriya, George Athanasopoulos, and Rob J. Hyndman. Optimal Forecast Reconciliation for Hierarchical and Grouped Time Series Through Trace Minimization. *Journal of the American Statistical Association*, 114:804–819, 2019.

[86] Jiangkai Wu, Yu Guan, Qi Mao, Yong Cui, Zongming Guo, and Xinggong Zhang. ZGaming: Zero-Latency 3D Cloud Gaming by Image Prediction. In *Proc. of ACM SIGCOMM*, pages 710–723, 2023.

[87] XPosed Developers. Xposed Framework API, 2023. https://api.xposed.info/reference/packages.html.

[88] Lingfeng Xu, Xun Guo, Yan Lu, Shipeng Li, Oscar C. Au, and Lu Fang. A Low Latency Cloud Gaming System Using Edge Preserved Image Homography. In *Proc. of IEEE ICME*, pages 1–6, 2014.

[89] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. PROTEUS: Network Performance Forecast for Real-Time, Interactive Mobile Applications. In *Proc. of ACM MobiSys*, pages 347–360, 2013.

[90] Jinyu Yang, Mingqi Gao, Zhe Li, Shang Gao, Fangjing Wang, and Feng Zheng. Track Anything: Segment Anything Meets Videos. *arXiv.2304.11968*, 2023.

[91] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proc. of ACM MobiCom*, pages 1–15, 2019.

[92] Zongxin Yang and Yi Yang. Decoupling Features in Hierarchical Propagation for Video Object Segmentation. *Advances in Neural Information Processing Systems*, 35:36324–36336, 2022.

[93] Roy D. Yates, Mehrnaz Tavan, Yi Hu, and Dipankar Raychaudhuri. Timely Cloud Gaming. In *Proc. of IEEE INFOCOM*, pages 1–9, 2017.

[94] Yuhan Zhou, Tingfeng Wang, Liying Wang, Nian Wen, Rui Han, Jing Wang, Chenglei Wu, Jiafeng Chen, Longwei Jiang, Shibo Wang, Honghao Liu, and Chenren Xu. AUGUR: Practical Mobile Multipath Transport Service for Low Tail Latency in Real-Time Streaming. In *Proc. of USENIX NSDI*, pages 1901–1916, 2024.