vSoC: Efficient and Debug-Friendly Virtual System-on-Chip for Mobile Emulation

Zijie Zhou, *Student Member, IEEE*, Jiaxing Qiu, *Student Member, IEEE*, Zhenhua Li, *Senior Member, IEEE*, Feng Qian, *Senior Member, IEEE*, Yunhao Liu, *Fellow, IEEE*, Bo Xiao, Hongwei Hu

***Abstract*—Emerging heavy-load mobile apps like UHD video and AR/VR access diverse high-throughput hardware devices, e.g., video codecs and cameras. However, today's mobile emulators exhibit poor performance when emulating these devices. We pinpoint the major reason to be the discrepancy between the guest's (system-on-chip) and host's (PC or cloud server) memory architectures for these devices, which makes the shared virtual memory (SVM) architecture of mobile emulators highly inefficient. To address this, we introduce vSoC, the first virtual mobile SoC featuring a *unified* SVM framework that enables efficient and secure data sharing among virtual devices, as well as an *intelligent* prefetch engine that effectively eliminates the vast majority of coherence maintenance overhead. While vSoC addresses runtime performance issues, app developers face a severe debugging challenge due to the inability of traditional tools to capture complete system states. Therefore, we devise an adaptive VM (virtual machine) snapshot-based approach that dynamically selects the optimal resource loading strategy to make vSoC debug-friendly. Compared to state-of-the-art emulators, vSoC brings 1.8–9.0× frame rates, 35%-62% lower motion-to-photon latency, and 6.5–14.4× bug reproduction rates for heavy-load apps. It is applicable to a variety of scenarios like end-user high-performance emulation and cloud/web-based rendering.**

***Index Terms*—virtualization, mobile emulation, system-on-chip, shared memory, coherence, snapshot, debugging, rendering.**

I. Introduction

MOBILE emulation enables mobile OSes/apps to execute seamlessly on PC or cloud server machines. Recently, mobile OSes/apps have been widely deployed on diverse platforms like smart cars and AR/VR headsets. Different from traditional mobile apps, emerging heavy-load apps on such platforms (e.g., UHD video and AR/VR) need much better performance (4-16× resolutions, 60-180 FPS, and sub-100 *ms* motion-to-photon latency) to deliver satisfying user experience [1]–[4]. Moreover, unlike heavy-3D apps that place a high load only on the GPU, heavy-load apps intensively interact with various high-throughput System-on-Chip (SoC) devices such as video codec, image signal processor (ISP), and camera. Thus, app developers have even stronger demands for mobile emulators that can emulate the entire mobile SoC efficiently, rather than individual devices in previous work [5]–[13].

Unfortunately, state-of-the-art (SOTA) emulators [8], [14]–[17] (including Trinity [8], a high-performance mobile emulator that can smoothly run GPU-intensive apps) exhibit poor performance when running heavy-load apps. They constantly suffer from performance issues like video stalls and high motion-to-photon latency, compared to execution on real mobile devices. With in-depth instrumentation of SOTA emulators in §III, we pinpoint the major reason to be *inefficient data sharing among virtual (peripheral) devices* due to the hardware architecture gap between mobile and PC/server systems—in a mobile SoC, devices efficiently share data through a unified memory architecture, while a PC/server typically uses per-device dedicated local memory and exchanges data with the main memory via buses.

To bridge this gap, mobile emulators have to adopt a shared virtual memory (SVM) architecture [18], presenting a unified address space to the guest OS by transparently maintaining *data coherence* across the distributed host memory, *i.e.,* devices sharing the same virtual memory should see the same data. Existing mobile emulators, influenced by PC/server virtualization's modular design [14], [15], treat virtual devices as independent entities, relying on the guest to enforce coherence (as demonstrated in Figure 1): a piece of the guest memory is mapped to system services and apps, with each device synchronizing its local copy.

Although the guest-centered SVM approach facilitates inter-device data transfer in principle, it wastes memory bandwidth due to frequent copying between devices and the guest memory. To make matters worse, mobile OSes and apps are built on the assumption that data sharing among devices is efficient on a mobile SoC, amplifying the inefficiencies in coherence protocols. The resulting latency spikes and unanticipated blocking during synchronization can severely degrade system responsiveness and user experience.

This paper presents vSoC, the first virtual mobile SoC that enables virtual devices to collaborate and share data efficiently. As shown in Figure 2, the key idea of vSoC is to break free from the traditional modular architecture of virtual devices by building a unified SVM framework. vSoC has a global view of virtual devices and their interactions, making it feasible to directly transfer data between virtual devices without guest involvement. Moreover, the unified framework enables capturing data dynamics in vSoC, fostering an efficient coherence protocol tailored to mobile emulation.

To design an efficient and secure coherence protocol, we perform in-depth measurement of shared memory in mobile systems. We observe that the predominant usage of shared memory is within *data pipelines*, which streamline data processing among SoC devices, using shared memory as intermediate storage. Another key observation is that since SoC devices have fast unified memory, the cross-device control and data flows of shared memory are usually ordered. To

Zijie Zhou, Jiaxing Qiu, Zhenhua Li, and Yunhao Liu are with Tsinghua University, Beijing 100190, China (zijiezhou017@outlook.com; jx.qiu@outlook.com; lizhenhua1983@gmail.com; yunhaoliu@gmail.com)

Feng Qian is with University of Southern California (fengqian@usc.edu).

Bo Xiao and Hongwei Hu are with Ant Group (xiaobo.xiao@antgroup.com; hongwei.huhw@antgroup.com).

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2026.3667012

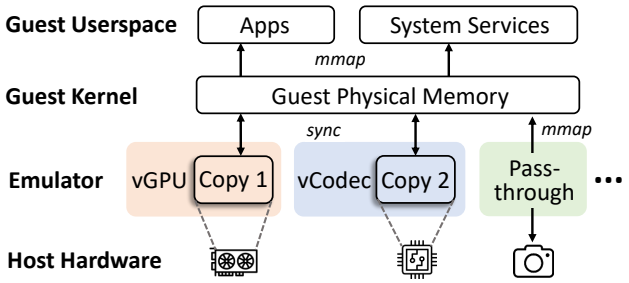


Figure 1: Memory architecture of a typical emulator.

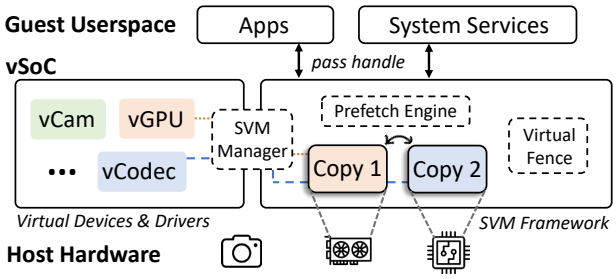


Figure 2: Memory architecture of vSoC.

ensure this, today's OSes employ various mechanisms such as buffering and VSync [19], which lead to unavoidable delays (averaging at 17 *ms*) between consecutive shared memory accesses. We refer to such delays as *slack intervals*.

While slack intervals are unavoidable, vSoC makes use of them through *intelligent prefetching*: it predicts when and where the next SVM access would occur, and fetches data updates to the predicted device ahead of time, thus hidding coherence maintenance under slack intervals. While prefetching has been widely used in computer systems, designing a prefetch engine for vSoC faces two unique challenges: how to robustly predict SVM accesses and how to accommodate prefetching to different slack interval durations.

To foster robust SVM access prediction, vSoC leverages its global vantage point to collect various SVM usage data, properly maintained as two-layer *twin hypergraphs*, and uses them to finely model the data flows of both virtual and physical devices. We use hypergraphs (where an edge can connect more than two vertices) because data flows in mobile systems may involve more than two devices. Leveraging the global data flow model, we demonstrate that even a simple algorithm (exponential smoothing [20]) can achieve a very high SVM access prediction accuracy of $\geq$99%.

To accommodate different slack interval durations, the prefetch engine carefully controls how guest device drivers should wait for a prefetch. Specifically, informed by the history of slack intervals and prefetch time provided by the hypergraphs, guest drivers adaptively compensate for the time difference if the slack interval is not long enough to cover the prefetch. This prevents the prefetch from blocking the next SVM access which hurts app performance. This idea of data prefetching also applies to a wide variety of relevant scenarios such as cloud- or web-based remote rendering, where reducing latency is essential for smooth end-user experience.

The SVM framework effectively addresses the runtime lag of heavy-load apps. During the deployment of vSoC (via Huawei DevEco Studio) as mentioned in our preliminary work [21], however, quite a few app developers reported a significant debugging challenge: unlike conventional apps, heavy-load apps exhibit complex states (and state dependencies), non-deterministic execution, and strict real-time requirements. Traditional debugging tools cannot capture complete cross-hardware states and in-situ rendering pipelines. This makes bug reproduction through logs or breakpoints ineffective or unreliable, leaving *VM (virtual machine) snapshots* as the only viable method to restore full system states. Unfortunately, existing snapshot frameworks suffer from long-lasting (>3 *s*) and severe (FPS=0) stalls after loading a heavy-load app from the snapshot. These stalls often mask underlying bugs such as GPU pipeline starvation or thread contention, making it difficult to isolate and reproduce them reliably. Therefore, we need to additionally design an efficient VM snapshot system to address this debugging challenge.

The snapshot restoration process involves two major phases: 1) deserializing resource data from snapshot files into memory, and 2) uploading them to the GPU. Existing snapshot frameworks only perform deserialization but defer GPU uploading until resource data are actually accessed by a rendering thread, *i.e.,* the so-called "lazy" uploading mechanism. For a heavy-load app, the enormous amount of graphics resource data may well make the uploading duration far beyond a single frame interval and thus cause severe stalls. A natural solution is to perform GPU uploading right after deserialization, which however would cause severe stalls instantly.

To effectively address the drawback, we carefully investigate the GPU uploading process, and discover that different resources are subject to distinct bottlenecks throughout the process, and thus should be handled with distinct uploading strategies among 1) synchronous uploading during snapshot restoration, 2) asynchronous uploading triggered by certain events, and 3) original lazy uploading. Therefore, the long-lasting uploading duration can be amortized across multiple phases and the whole process can be finished before the resource data are actually accessed by a rendering thread.

Specifically, we devise an adaptive VM snapshot restoration approach that dynamically selects the optimal uploading strategy for each resource based on its unique characteristics. Since static properties (*e.g.,* size and format) of a resource fail to reflect dynamic behavioral patterns like usage frequency and inter-resource dependencies, we resort to continuously monitoring the activities of rendering threads, as well as utilizing a self-tuning regression model that adaptively adjusts coefficients through runtime feedback to quickly figure out the best uploading strategy for each resource in real time.

We integrate the above designs and implement vSoC for Android and OpenHarmony, two Linux-based mobile systems, in 101K lines of C/C++ code. We compare its performance against five mainstream mobile emulators (Google Android Emulator [14], QEMU-KVM [15], LDPlayer [17], Bluestacks [16], and Trinity [8]) with SVM microbenchmarks, top-25 popular mobile apps, and 50 heavy-load apps. Compared to other emulators, vSoC brings 12%-49% higher FPS to the top

popular apps, while achieving 1.8–9.0× FPS, 35%–62% lower motion-to-photon latency, and 6.5–14.4× bug reproduction rates for heavy-load apps.

In summary, the paper makes the following contributions:

- We present vSoC, the first virtual mobile SoC that breaks free from the classical modular architecture and mitigates inefficient memory sharing between virtual devices.
- We implement a novel VM-snapshot framework and demonstrate its capability to significantly improve bug reproduction rates, making vSoC debug-frendly.
- We implement vSoC for two mainstream open-source mobile systems and show that it can achieve considerable performance improvements on a variety of applications.

The remainder of this paper is organized as follows. §III introduces the background and motivation, §IV presents the basic design of vSoC, and §V describes our debug-friendly efforts for vSoC. Afterwards, we detail the system implementation in §VI and evaluate its performance in §VII. Finally, §II discusses the related work and §VIII concludes the paper.

II. Related Work

Shared Virtual Memory. SVM systems are essential for enabling seamless data sharing across different hardware architectures. Current SVM research focuses on three main aspects: operating systems support [22]–[28] that designs robust memory management strategies for heterogeneous memory systems, compiler enhancements [29], [30] that provide transparent SVM implementations for programs, and hardware integration [31]–[34] that provides low-level support for SVM. Nevertheless, they are mainly designed for scenarios like distributed storage or parallel computing, and thus are not directly applicable to mobile emulation or emerging scenarios like cloud- and web-based graphic rendering (see §III-B).

Moreover, regarding the choice of coherence protocols, many SVM implementations adopt classical write-invalidate [35] or broadcast [36], [37] protocols, which are unsuitable in mobile emulation because of high overhead. Some works explore software [38] and hardware prefetching [26], [27] for coherence management, but typically face a tradeoff of prefetch aggressiveness. Regarding the granularity of SVM memory units, most SVM related work [18], [26], [27], [33] adopts a fixed page-level unit size, whereas mainstream mobile emulators often segment SVM into smaller memory units, because processing large numbers of page faults across the virtualization boundary can bring significant VM-Exit [8], [39] overhead that stalls the whole system.

Snapshot and Live Migration. Snapshot and live migration represent two fundamental techniques for capturing and transferring virtual machine states. While snapshots preserve system states at specific checkpoints for later restoration, live migration enables continuous operation transfer between hosts with minimal downtime. The research community has primarily focused on optimizing live migration [40]–[45], driven by its critical role in secure cloud computing. Advancements have achieved progress in reducing migration downtime [40] and optimizing memory page transfer [41], making live migration a mature solution for stateless services [46] [47] [48] [49] and homogeneous VM transfers.

However, these solutions exhibit inherent limitations when applied to graphics-intensive mobile emulation. Traditional migration scenarios typically involve server applications with simple display outputs (e.g., terminal interfaces or remote desktop protocols), where GPU state preservation is either unnecessary or handled through auxiliary protocols like VNC [50]–[52]. In contrast, mobile emulation requires precise reconstruction of complex GPU pipelines (textures, shaders, framebuffers) and real-time rendering states. This fundamental difference explains the scarcity of research addressing snapshot/migration for graphics-heavy emulation environments, creating the gap our work aims to fill.

III. Background and Motivation

The inefficiency of data sharing among virtual devices is not mere coincidence; it originates from the hardware architecture gap between mobile and PC/server systems. In this section, we introduce the shared memory abstraction in mobile systems (§III-A), how the abstraction is implemented in typical emulators (§III-B), reveal the real-world characteristics of shared memory (§III-C), and finally, discuss its implications for the design of vSoC (§III-D).

A. Mobile SoC and Shared Memory

The shared memory abstraction in mobile systems arises from the pursuit of efficiency. To make mobile platforms power- and space-efficient, their hardware universally adopts the SoC architecture [53], where multiple devices (*e.g.,* CPU, GPU, ISP, and camera) are packaged into one chip and linked to a single physical memory with a fast interconnect.

Consequently, the architecture of SoCs has shaped how mobile systems interact with them. Since SoC devices physically share memory, mobile systems provide shared memory interfaces to ease data management and sharing across devices, for instance the `AHardwareBuffer` interface [54] in Android, and the `OH_NativeBuffer` interface [55] in OpenHarmony. The interface sits at the Hardware Abstraction Layer (HAL) [56] of a mobile system, which is typically called by system services or apps and implemented by SoC manufacturers along with other device drivers.

A shared memory interface typically consists of the APIs like alloc and free, providing a handle-based representation of shared memory. Since SoC devices adopt a unified memory architecture, the actual allocation takes place in the shared physical memory. The virtual address of the corresponding region can be obtained and accessed by CPU by calling `begin_access`. Moreover, the shared memory interface is usually deeply integrated with the interfaces for other SoC devices (*e.g.,* OpenGL ES [57], OpenMAX [58], Camera HAL [59]), so the handles can be directly passed to other SoC devices. In this way, data can be shared among mobile SoC devices efficiently.

B. Shared Virtual Memory in Mobile Emulation

While a mobile SoC achieves high power and space efficiency through a monolithic architecture, PC/server devices are

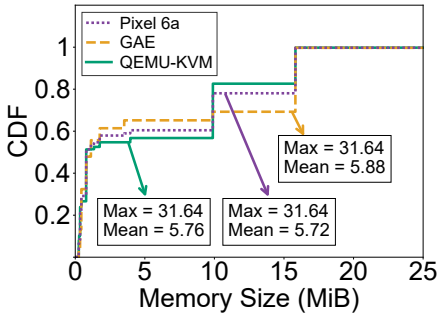


Figure 3: Size of shared memory.

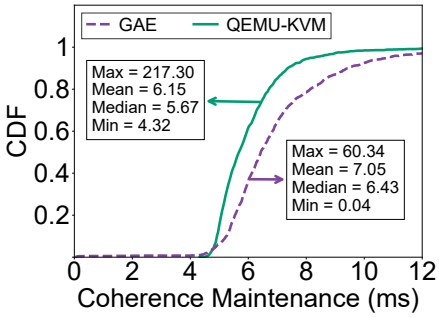


Figure 4: Coherence time cost.

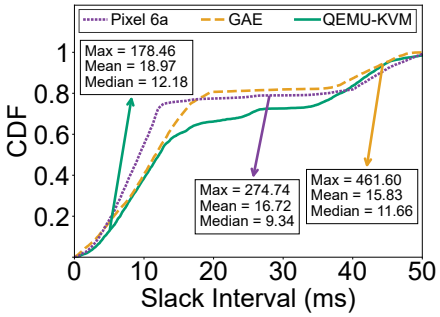


Figure 5: Slack intervals.

usually modular: they are connected to the main memory via buses like PCI-e [60], allowing for upgrades and replacements in case of failure. Since they are farther from the main memory, many of them (*e.g.,* GPUs and DSPs) are equipped with dedicated device memory to accelerate local processing.

To support the mobile shared memory abstraction on PC/server devices, mobile emulators have to adopt an SVM architecture [18], which presents the illusion of a unified address space with physically distributed memory. The illusion is achieved by allowing PC/server devices to have local copies of the SVM data and copying data between devices to make sure that they access up-to-date data.

However, existing mobile emulators heavily follow the modular PC/server hardware architecture, making SVM coherence maintenance a key challenge towards the goal of efficient data sharing. In the emulators, virtual devices are designed to operate independently of each other; they can even use different I/O virtualization techniques, like a paravirtualized GPU and a passthrough camera. Even for Google Android Emulator [14] which is optimized for mobile systems, we only observe limited collaboration between the virtual codec and the GPU device.

Since virtual devices in mobile emulators are largely isolated from each other, SVM coherence is typically maintained with the help of guest memory. For each SVM region, a piece of guest memory is allocated through `kmalloc` [61] and mapped to userspace via `mmap` [62], so that mobile system services and apps have the same view of the guest memory. Each virtual device only needs to keep the guest memory up to date, by fetching data to and from its local memory[1].

Nevertheless, this memory architecture can lead to high memory bandwidth overhead when shared memory is used as *intermediate storage* between two devices, *i.e.,* SVM data are written by one device and read by another. Taking the previous example (P_a writes and P_b reads), after P_a writes, V_a copies the written data to the guest memory G, and before P_b reads, V_b copies the data from G back to P_b. For a simple pair of W/R operations, data have to be copied twice (to and from the guest), not to mention that they cross the virtualization boundary which further slows down the copying. Worse still, such usage of shared memory is frequent in today's heavy-load mobile apps, as will be revealed below.

[1]Local memory might be device memory for hardware-accelerated virtual devices, or main memory for software-emulated devices.

Table I: The five types of heavy-load apps involved.

Type	Devices Involved	Count	Duration
UHD Video	Codec, GPU, Display	10	5 min per app
360° Video	Codec, GPU, Display	10	5 min per app
Camera	Camera, ISP, GPU, Display	10	5 min per app
AR	Camera, ISP, GPU, Display	10	5 min per app
Livestream	Codec, GPU, Display, NIC	10	5 min per app

C. Real-World Usage of Shared Memory

To understand the real-world usage of shared memory in mobile systems, we perform in-depth measurements of shared memory in Android.

Workloads. As shown in Table I, we choose 50 heavy-load apps from five categories that run at high visual fidelity (UHD + 60 FPS) and simultaneously use multiple SoC devices. For the first three categories, we select the top-10 popular apps from Google Play. For AR apps, since the Google ARCore [63] framework is not supported on most emulators (but is adopted by many AR apps), we select the top-10 popular apps that can run without Google ARCore. For livestream apps, we select the top-10 popular apps that support video streaming over local area networks to minimize the impact of network instabilities on the results.

To keep the comparisons fair and straightforward, we strictly control various aspects of the workloads. The videos played in the UHD/360° video apps are of UHD resolution (3840×2160), with 60 FPS frame rate and 300 Mbps bitrate. Livestream apps are configured to use RTMP [64], a universally supported livestream protocol. The streaming resolution is set to UHD, the frame rate is set to 60 FPS, while the bitrate is left default since different apps support different bitrate ranges. We serve livestream requests using nginx [65], a popular web server program. The web server runs on a dedicated machine with Intel i7-8700K CPU and 32 GB DDR4 memory running Ubuntu 22.04, connected to the devices under test using Gigabit Ethernet.

Devices Under Test. The measurement is conducted on a Google Pixel 6a device and two open-source emulators: Google Android Emulator (GAE) [14] and QEMU-KVM [15]. The basic configurations of the emulators are the same as those of the physical device, each with an 8-core CPU, 6 GB memory, and a Full-HD+ (2400×1080) display. The emulators run on a high-end commodity desktop PC with a 24-core Intel i9-13900K CPU @ 3.0 GHz, 64 GB RAM (DDR5 5600 MHz), a NVIDIA RTX 3060 dedicated GPU, and a HIKVISION V148 USB camera capable of streaming UHD video at 60

FPS. GAE is run on the Windows 11 23H2 version, and QEMU-KVM is run on Ubuntu 22.04.

Methodology. We instrument the test system to obtain detailed traces of SVM usage. More concretely, we instrument the shared memory interface (§III-A) to collect detailed information of shared memory including its size, R/W usage, and API call duration, as well as the name of the caller process/thread to identify the app or system service using the interface[2]. For the emulators, we further instrument their SVM implementations to track the usages of SVM in virtual devices and the durations of coherence maintenances. That is why we choose open-source emulators instead of commercial emulators like Bluestacks [16] in the measurement.

Observations. Shared memory is frequently used in all the heavy-load apps, with an average of 261-323 API calls per second for each category. The top-3 system services / apps that heavily use the shared memory are all hardware-related: media service (28%, operates the codec device), SurfaceFlinger (23%, operates the GPU), and camera service (19%, operates the camera and ISP).

Interestingly, the vast majority (99%) of SVM regions only serve one or two processes, and further, 96% of SVM usages in these regions exhibit a regular cyclic R/W pattern: after the first process writes to SVM, the second process reads the data, and then the first process writes again, and so on. The regular pattern suggests that most SVM regions are part of one-way *data pipelines*. Like an image pipeline in camera apps where the camera captures, the ISP processes, and the GPU renders the image, data pipelines boost the power efficiency of mobile platforms by streamlining data processing among specialized SoC devices. In a pipeline, shared memory is used as intermediate storage to forward data between devices.

We also observe minor usages of the shared memory interface. For instance, a minor portion of shared memory accesses (1%) exclusively happen between app processes and are only accessed by CPU, indicating that the shared memory interface might be used for normal IPC as well.

Moreover, as shown in Figure 3, the shared memory regions allocated by the apps are usually quite large (49% of regions are more than 1 MiB). On all three platforms, there are two prevalent sizes of shared memory: 9.9 MiB and 15.8 MiB. We discover that they are respectively the size of display buffers (9.9 MiB = 2400×1080×4 Byte) and UHD video frames (15.8 MiB = 3840×2160×2 Byte), further demonstrating the prevalence of data pipelines. The large size of shared memory also leads to high coherence maintenance overheads in the emulators. As shown in Figure 4, the average duration of coherence maintenance in GAE and QEMU-KVM are as high as 7.1 *ms* and 6.2 *ms*.

More importantly, we observe that mobile systems typically use shared memory in an ordered but uncontinuous way. SVM accesses do not happen immediately next to each other; instead, there are intervals between adjacent accesses that happen in two processes, which we term as *slack intervals*. As shown in Figure 5, the slack intervals are typically tens of milliseconds (avg. 17.2 *ms*), usually longer than the coherence maintenances of the emulators (avg. 6.7 *ms*). Slack intervals exist because it is hard to pace the execution of devices with millisecond-level accuracy even for a physical SoC. For instance, many system services adopt access synchronization mechanisms (*e.g.,* VSync [19]) to protect shared memory from concurrent writes. Some latency-insensitive pipelines (*e.g.,* video playback pipelines) further use buffering to smooth out jitters, so in Figure 5, some slack intervals (>30 *ms*) are significantly longer than others (<20 *ms*). It is worth noting that these OS-level synchronization mechanisms (*e.g.,* VSync and buffering) are independent of the underlying hardware, so the durations of slack intervals on emulators and the physical device are very similar.

[2]In Android, system services typically live in independent userspace processes to improve system security and stability.

D. Implications for vSoC

The measurement of real-world SVM usage provides us with valuable insights into the design of vSoC.

A key performance issue in existing emulators is slow coherence maintenance, where each data pipeline involves one or more coherence operations taking <6 *ms* due to inefficient data copies across the virtualization boundary (see §III-B). This significantly impacts performance, as apps running at 60 FPS have only 16.7 *ms* per frame, leaving limited time for actual data processing and often causing slow or dropped frames. To address this, we propose leveraging slack intervals in mobile systems through a prefetch coherence protocol, which tracks SVM data flows, predicts the next accessing device based on historical usage, and prefetches data updates during slack intervals. This approach hides coherence maintenance under slack intervals, drastically reducing time overhead. While our design targets heavy-load mobile app emulation, the same principles of unified data sharing and predictive prefetching are also applicable to other scenarios like cloud/web rendering.

The prefetch protocol, nevertheless, comes with obstacles in reality. Prefetch requires coordination between multiple virtual devices, which is hard to achieve with typical emulators whose virtual devices have limited knowledge of each other (§III-B). Furthermore, without access to enough cross-device SVM usage information, prefetching can suffer from frequent prediction failures which completely nullify its benefits. Whenever a prediction failure happens, coherence maintenance has to be re-performed, leading to high overhead in both time and bandwidth, while additionally wasting the ahead-of-time data copies. Therefore, to enable efficient coherence management of SVM, a unified architecture for the shared memory is necessary.

IV. Basic Design of vSoC

This section presents the basic design of vSoC, which aims to address the runtime performance issues of existing mobile emulators. §IV-A introduces the overall architecture of vSoC, while §IV-B, §IV-C and §IV-D detail the design choices made in various components of vSoC architecture.

A. Design Overview

vSoC is designed to achieve four key objectives: (1) providing unified shared memory virtualization across virtual devices; (2) developing an efficient coherence protocol optimized for mobile emulation; (3) addressing performance limitations in existing SVM architectures; Our solution builds on a paravirtualized SVM framework (Figure 2) that maintains compatibility with standard virtualization techniques while optimizing for device development scenarios requiring guest-host resource sharing. The framework implements a `virtio` [66] based architecture that preserves the guest kernel while enabling efficient cross-device communication.

`virtio` is a para-virtualized device framework that accelerates device I/O performance by forwarding commands and data from guest drivers to real host hardware. Because of its efficiency and flexibility, `virtio` is widely regarded as a de facto standard and a practical foundation for peripheral emulation in typical virtualization scenarios. Many mainstream emulators, such as GAE [67], QEMU [15], BlueStacks [16], and Trinity [8] [68], build their own virtual devices on top of `virtio`. On the other hand, each `virtio` device has to maintain its own independent virtqueue and driver logic, a design that inherently lacks native support for peer-to-peer data coherence between virtual devices. This architectural limitation prevents efficient cross-device data exchange and synchronization. As a result, heavy-load applications are often bottlenecked by `virtio`'s *isolated device model* (despite being more efficient and secure than its classic *split device model*), leading to significant runtime performance degradation. To address this, our SVM framework is built on top of the `virtio` drivers as a complementary component that enables efficient data coherence and unified memory management across devices.

For security concerns, our shared memory framework strictly adheres to the simulated mobile SoC hardware permission model. Its core contribution lies in optimizing data transmission paths and timing without altering data access authorization rules, which continue to be strictly enforced by the simulated hardware and the Guest OS. As a consequence, this design neither compromises the isolation between independent hardware components nor introduces additional security risks.

In summary, the framework comprises three core components: the SVM Manager (§IV-B) for unified memory representation, the Prefetch Engine (§IV-C) for efficient coherence, and the Virtual Command Fence (§IV-D) for low-overhead synchronization.

B. SVM Manager

The SVM Manager implements mobile systems' shared memory interface (§III-A), managing SVM resource lifecycles for vSoC. Each SVM region receives a unique 64-bit ID upon allocation, with memory space lazily allocated since the accessing device is only known at first access. To maintain guest-host isolation, most SVM operations must be performed by the host. The guest caches only essential metadata (e.g., size) for control operations, while complete metadata and resource handles (e.g., guest memory scatterlists or GPU memory handles) reside in a host-side hashtable.

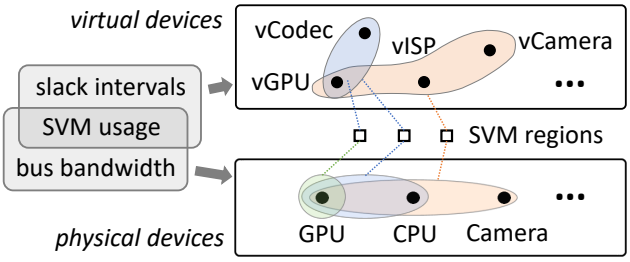


Figure 6: The structure of the twin hypergraphs. Each hyperedge characterizes a data flow. The dotted lines linking the two hypergraphs represent hashtable entries. The shaded areas to the left exemplify the data recorded in each hypergraph.

This unified representation allows vSoC's virtual devices/drivers to reference SVM regions by ID, eliminating data transfer in device commands and coherence overhead. When shared memory transfers data between devices, coherence is maintained directly among virtual devices without guest intervention, avoiding extra bandwidth consumption (§III-B).

Through interacting with the virtual devices, SVM Manager collects SVM-related statistics across the system stack. As will be detailed in the next subsection (§IV-C), such global information plays a key role in enabling an efficient and robust coherence protocol. To be concrete, SVM usage is collected with a two-layer graph structure termed *twin hypergraphs*. As shown in Figure 6, the twin hypergraphs consist of two directed hypergraphs that respectively model the data flows of virtual and physical devices, and a hashtable in between to map the SVM regions to the data flows in the two hypergraphs. The twin hypergraphs are maintained in the host and initialized at emulator startup. The nodes of the hypergraphs respectively represent virtual and physical devices and are known at compile time, while the hyperedges as well as the hashtable mappings are dynamically constructed at run time.

Essentially, the data flow of an SVM region results from data dependency: data that were previously written by one device are now read by another. A data flow can be described by its source and destination devices, whose relationship can be captured by a directed edge pointing from the source device to the destination. In mobile systems, we use hyperedges when recording data flows because data dependency might involve more than two devices, for instance when a write in camera is accompanied by two reads in ISP and GPU. Note that data flows and SVM regions have a one-to-many relationship: different SVM regions might have the same data dependency (*e.g.,* when the data pipeline enables buffering and a chain of buffers correspond to multiple SVM regions), so they are all characterized by one hyperedge.

The need for two separate hypergraphs for virtual and PC/server devices arises from the fact that virtual devices do not have a one-to-one relationship with the underlying hardware. A virtual device can dynamically map to the most appropriate physical device depending on the guest workloads. For instance, when the guest requires decoding of a video format the underlying codec or GPU device does not support,

we have to fall back to software decoding by CPU. On the other side, multiple virtual devices can utilize the same physical device as well. As an example, although GPUs and displays are two discrete SoC modules that manage their own resources and we need to provide two distinct virtual devices for them, displays are usually managed by GPUs in the PC/server host, in which case virtual displays ultimately interact with the physical GPU as well.

The primary use of the twin hypergraphs is to group SVM regions into data flows and record both high and low-level statistics of data flows with its two layers. Since virtual devices directly interact with the guest, high-level information of each data flow is recorded in its hyperedge, *e.g.,* the virtual devices using the SVM, and the slack intervals between consecutive cross-device SVM accesses. The physical layer, in contrast, records low-level properties related to the actual data transfer, including data size and the available bandwidth of physical devices. With the twin hypergraphs, data flows across the entire virtual SoC are captured, but the recorded information in each hypergraph is partial. Therefore, we use a hashtable in between to map the SVM regions to the hyperedges in the two layers. The mappings are dynamically updated when SVM accesses are processed by the SVM Manager.

C. Prefetch Engine

While the SVM Manager decides *what* to copy in coherence maintenance, the prefetch engine deals with *when*. The prefetch protocol has been briefly described in §III-D: it overlaps coherence maintenance with the slack intervals. In reality, however, prefetch has to be carried out with care to avoid problems that seriously affect its performance. Firstly, since the protocol needs to warm up with historical SVM usage, predictions usually fail for newly allocated SVM regions, incurring high performance penalties on apps that frequently switch data pipelines (*e.g.,* short-form videos).

More importantly, in 26% of the measurement cases, the slack intervals are insufficient to cover coherence maintenance; the next SVM access will have to wait until the prefetch completes, leading to high SVM *access latency*. Unfortunately, even a slightly longer SVM access latency (*e.g.,* 2 *ms*) can create a serious chain reaction in mobile system services and apps. More concretely, high SVM access latency causes apps to miss the current frame deadline and wait for the next (>16 ms of waiting), and lead to visible frame drops.

The above problems set the following design goals for a satisfying coherence protocol for vSoC: (1) maximizing the accuracy of the predictions made in the protocol, (2) minimizing the access latency of the next SVM operation to avoid the chain reaction, and (3) maximizing the overlap between coherence maintenances and the slack intervals.

To meet the goals, we design a prefetch engine that adaptively adjusts the *synchronism* of the prefetch protocol. The basic idea is to control how guest drivers should wait for the prefetch (*i.e.,* whether guest and host executions should be synchronous). If the slack intervals are not long enough to cover prefetch, guest drivers can compensate for the time delta by blocking ahead of time, as if the prefetch operation

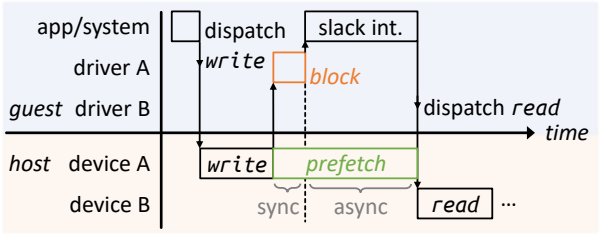


Figure 7: Timeline of the robust prefetch protocol.

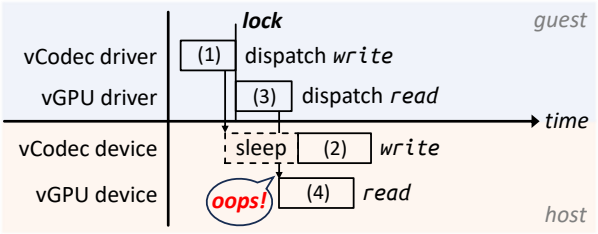
is done synchronously. When the guest driver finishes the compensation, it proceeds to other tasks (such as notifying the completion of the SVM operation), and the remaining portion of prefetch is done asynchronously.

Figure 7 exemplifies how the prefetch engine works. The app/system commands device A to write to an SVM region and then commands device B to read from it. Prefetch begins immediately after device A finishes writing. Then, to avoid SVM access latency, A's driver blocks for the 2 *ms* time delta before returning the control flow of the SVM to the system, making the prefetch operation synchronous and blocking for the first 2 *ms*. If driver A does not block, when the next read is dispatched, driver B will have to wait for 2 *ms* (until the prefetch completes) before it can access the actual data. However, such access latency is unexpected to the app/system and will disrupt their schedule.

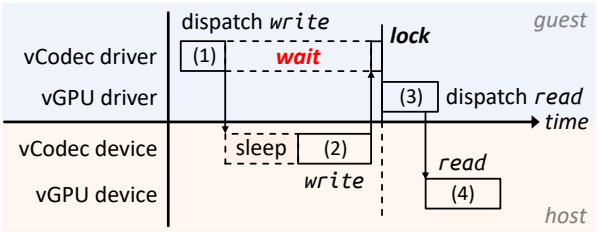
To realize the robust prefetch protocol, some of the statistics need to be predicted, and that is where the twin hypergraphs come in. In general, two types of predictions are involved in prefetching. The first type is data dependency—whether a read will take place after a write, and which physical device will perform the read. To this purpose, we utilize the R/W history of the physical data flow associated with an SVM region, including the physical device ID and the operation type. Within the same data pipeline, the prediction accuracy can reach 99+%, as the R/W operation flow shows very ordered patterns. We record R/W history into coarse-grained data flows instead of fine-grained SVM regions to achieve zero-shot predictions for new SVM regions when switching data pipelines.

The second type of prediction is on how much time the guest driver needs to compensate for the prefetch. The following statistics are used: (1) size of the dirty SVM region; (2) available bus bandwidth between two physical devices; (3) historical slack intervals between virtual devices. Predictions of this type can also be accurate because the statistics are usually stable throughout the lifetime of a data pipeline.

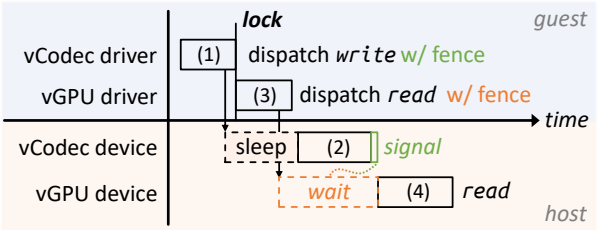
Regarding the actual prediction algorithms, since both slack intervals and bus bandwidths can be seen as univariate time series and exhibit no clear trend or seasonality patterns, we use the single exponential smoothing algorithm [20], one of the most widely used forecasting techniques due to its simplicity, robustness, and accuracy [69]. In the algorithm, the predicted value is a weighted average of the past values, and the weights decay exponentially over time according to a hyperparameter α. α is empirically chosen as 0.5 according to our benchmarks. This idea can be further generalized to a wider range. For example, web-based cloud rendering [70]



(a) Guest-side locks do not protect host execution



(b) Guest-side lock & wait block the guest driver



(c) Guest-side lock & fence achieve efficient access ordering

Figure 8: An example of access ordering between virtual devices.

[71] has recently emerged as an important building block for securing the remote working environments of enterprises. In such a scenario, predictive data delivery can hide network and decoding delays in much the same way as our prefetch protocol hides inter-device coherence latency.

D. Virtual Command Fence

Like mobile emulators [8], [14], vSoC uses asynchronous communication between guest drivers and host devices. Commands are batched through device-specific queues to minimize virtualization overhead. However, this asynchronous model creates ordering challenges when multiple devices access shared memory. Take the common case of video rendering as an example. The ideal execution order is as follows: (1) The codec driver dispatches a `write` command to the virtual codec device to write a decoded video frame to an SVM region; (2) The virtual codec device writes to the SVM; (3) The guest OS forwards the SVM handle to the GPU driver; (4) The GPU driver dispatches a `read` command to the virtual GPU device to read from the SVM and start drawing. Steps (1) and (3) happen in the guest and are protected by locks, but if the emulator does not synchronize steps (2) and (4), out-of-order execution may happen (as shown in Figure 8a).

To solve the problem, a common approach [8], [14] is to let the guest driver perform shared resource operations *atomically*. As shown in Figure 8b, in step (2), the codec driver waits until the virtual codec device finishes writing (hence "atomic" write) before returning the SVM handle to the guest OS. Nevertheless, atomic operations introduce a head-of-queue blocking problem: time-consuming atomic operations block the codec driver from processing subsequent commands and reduce overall throughput. Another approach is to adopt an event-driven paradigm, where the guest driver dispatches the instruction and proceeds to other tasks, until the host execution has finished and notifies the guest driver via emulated interrupts. The approach avoids guest idling, but incurs additional VM exits triggered by interrupt notifications.

In essence, the key challenge to efficient access ordering is that command orders are only known to guest drivers, but need to be enforced in the host. Existing paradigms all rely on guest drivers to guarantee access ordering, leading to frequent guest-host control flow synchronizations that bring high overhead. Therefore, we propose an alternative host access ordering mechanism with minimal guest involvement, termed *virtual command fences*.

The core idea is to attach *virtualized* instances of fences to the commands dispatched by the guest, so that order semantics can be carried to and enforced by the host alone. There are two types of virtual fence instructions: one that "signals" when the preceding operations have finished, and one that "waits" until the signaling happens. They are typically used in pairs to represent a *happens-before* relationship [72], though multiple waits on a "signal" fence are also allowed. Figure 8c illustrates an example usage of the fences. The "wait" fence acts like a barrier and ensures that the `read` command will only be executed after the associated `write` has completed. Meanwhile, the guest drivers are not affected and can process subsequent commands.

Virtual command fences are designed to provide a unified abstraction of access ordering for guest drivers, so that they do not have to worry about intricate details regarding how access ordering is done in the host. In fact, PC/server devices like GPUs are asynchronous from CPU in nature, so when guest commands involve execution in such devices, the host needs to use device-specific synchronization primitives (*e.g.,* `glFenceSync` for GPUs) to make sure that commands sent to those devices are completed as well. Therefore, we maintain a set of physical fence tables that track the status of device-specific synchronization primitives of each PC/server device, and a virtual fence table that aggregates the statuses and facilitates status queries. To minimize the overhead incurred by status queries, the virtual fence table is stored in the guest kernel and shared to the host via memory-mapped I/O (MMIO), while the physical fence tables are stored in the host.

V. Debugging Challenge and Resolution

This section presents vSoC's solution to the critical debugging challenges through VM snapshots. §V-A motivates snapshots for debugging under heavy loads. §V-B identifies fundamental design flaws in existing snapshot systems. §V-C examines performance bottlenecks and potential solutions. And finally, §V-D presents vSoC's adaptive snapshot design.

A. Background and Motivation

While vSoC effectively addresses runtime lag in heavy-load apps, a key requirement for debugging under heavy-load conditions remains: VM snapshots. Snapshots are complete, atomic captures and restorations of a virtual machine's state, including CPU registers, memory contents, GPU pipeline states, *etc.* They allow developers to save execution states at critical program points (e.g., before a rendering frame or during GPU pipeline stalls) and repeatedly restore them for deterministic debugging. When using mobile emulators for app development, traditional debugging tools like LLDB breakpoints [73], Android Debug Bridge (ADB) [74], and Logcat [75] provide essential capabilities like monitoring runtime logs and inspecting UI hierarchies.

To diagnose bugs in heavy-load apps, developers must capture complete system states that simultaneously reflect all hardware components. However, these apps exhibit three defining characteristics that render traditional debugging approaches unable to meet this requirement.

First, they demonstrate complex state dependencies. A single rendering frame's correctness often hinges on hundreds of interdependent variables such as active GPU pipeline states and pending memory transactions. Conventional debuggers cannot capture these distributed states atomically, as they inspect system components (*e.g.,* CPU registers and PU pipelines) individually. Snapshots handle this by first pausing the system to preserve dependencies, then systematically saving each component in its exact interlocked state. Second, these apps exhibit non-determinism where missing or incorrect locks/barriers create time sensitive bugs. Conventional debuggers mask these defects by perturbing the very contention patterns that reveal them, while snapshots preserve the exact faulty contention states. Third, heavy-load apps demand µs-level timing precision across critical execution paths. Traditional debuggers disrupt timing with their probing, while snapshots freeze the entire system in a single atomic operation.

Due to these constraints, VM snapshots remain the only viable solution for capturing and restoring complete system states in heavy-load environments. However, current implementations exhibit critical performance limitations: snapshot restoration typically induces extended system stalls (>3 *s*) and complete rendering pauses (FPS=0) when reloading heavy-load apps. More importantly, these artificial stalls frequently conceal the bugs they aim to expose *e.g.,* GPU pipeline starvation and thread contention patterns, making it difficult to isolate and reproduce bugs reliably.

When designing the optimized snapshot framework, we focus on the loading phase for two reasons: (1) the capture phase follows a simple and rigid sequence with very limited potential for further optimization; and (2) it is executed asynchronously without blocking the user workflow, thus having little impact on user experience. Given the deterministic nature and trivial runtime impact of the capture phase, most performance-related feedback we have received about snapshots focuses exclusively on the loading phase, which directly affects bug reproduction and debugging. Our optimization efforts therefore target snapshot loading, the actual bottleneck in both performance and user experience.

Table II: GPU resource upload performance in OpenGL

Loading Scenario	Resource Size	GPU Upload Time
lightweight single-frame update	1-2 *MB*	8-12 *ms*
heavy-load single-frame update	100-200 *MB*	800-1200 *ms*

B. Existing Snapshot Mechanism

Among existing Android emulators, only Google Android Emulator (GAE) supports full snapshots (commercial solutions like BlueStacks/LD Player are game-oriented and thus have limited debug support, while QEMU-KVM/Trinity lack GPU snapshot functionality). Initially, we adapted and reused GAE's open-source snapshot framework. However, our evaluation revealed that while GAE's snapshot mechanism handles lightweight apps effectively, it exhibits severe performance degradation when processing heavy-load apps. Specifically, a lightweight app is one in which all peripheral devices incur pretty light load. In fact, lightweight apps, heavy-3D apps, and heavy-load apps form a nested hierarchy of emulator-support difficulty, ranging from the easiest (lightweight) to the most demanding (heavy-load). The unified SVM framework introduced in the previous sections advances runtime performance from supporting heavy-3D apps to smoothly supporting heavy-load apps, whereas the existing snapshot framework can only handle the most basic lightweight apps. Based on our experimental setup using the high-end PC in Section III-C, we conducted fine-grained instrumentation and meticulous design analysis of GAE's snapshot framework. Below we elaborate the findings.

Restoring graphics resources during snapshot loading involves two critical steps. First, deserializing resource data from the in-storage snapshot file into memory. During snapshot creation, the emulator serializes all graphics resource metadata (e.g., texture dimensions) along with raw pixel data and saves it to disk storage. When loading snapshots, the system must first deserialize the data back into memory. Our measurements show this process completes relatively quickly (averaging about 10 *ms* for lightweight apps and about 1000 *ms* for heavy-load apps), compared to the total snapshot loading latency (averaging 800-1200 *ms*, including CPU and memory state restoration), this deserialization phase constitutes a negligible portion and has already been highly optimized through modern storage stack optimizations.

The second step is uploading the resources to the GPU. Take OpenGL texture data as an example. After reconstructing their metadata (width, height, *etc.*) and the actual pixel content in memory, the system needs to execute standard OpenGL API calls (*e.g.,* glTexImage2D) to transfer these resources to the GPU. This upload phase proves significantly more time-consuming than memory deserialization, and its performance is influenced by multiple factors. The timing and methods of triggering these uploads offer various implementation choices, presenting substantial opportunities for optimization.

Through in-depth analysis of the source code, we discovered that existing snapshot frameworks employ a highly passive lazy loading strategy. Specifically, they only perform the first step during loading. This process, known as lazy loading,

works as follows: Each graphics resource is associated with a *needrestore* flag, which is set to *true* after the snapshot is loaded. When a rendering thread attempts to access a resource that hasn't been uploaded yet, the thread is paused, and the system performs a synchronous resource upload (blocking until completion) before resuming execution.

While lazy loading works well for lightweight apps, it leads to severe performance degradation with heavy-load apps. To understand this issue, we conduct a detailed quantitative analysis of the underlying mechanisms. First, it is important to recognize that the main thread responsible for final rendering in mobile operating systems does not render continuously but rather at regular intervals, creating what we have referred to in previous chapters as the slack interval (averaging about 17 *ms*). Table II shows a comparison of typical graphics resource upload scenarios during a single frame interval for these two types of apps. We can observe that for lightweight apps, each lazy loading event requires an upload time of 8-12 *ms*, which falls within the frame interval and thus does not disrupt the main thread's rendering cadence. In contrast, for heavy-load apps, the upload duration significantly exceeds a single frame interval (e.g., 800-1200 *ms* vs. $<$17 *ms*).

The 800-1200 *ms* data transfer already exceeds slack interval tolerance. Worse still, these compounding effects amplify latency to $>$3 *s* stalls. This is because the latency fundamentally disrupts the rendering pipeline's normal timing. Such long-lasting starvation forces GPU work queues to reset their scheduling patterns, while driver timeout mechanisms trigger redundant state revalidation. These secondary effects compound the initial delay, turning milliseconds into seconds.

This analysis uncovers that for heavy-load apps, the observed stalls are primarily caused by the mismatch between existing coarse-grained lazy loading mechanisms and the complexity of heavy-load apps' graphics resources. With the increasing demand for heavy-load app development, existing snapshot frameworks have proven inadequate, and the current lazy loading strategy is no longer viable. Consequently, snapshot loading requires a new, more adaptive design to address these challenges effectively.

C. Resource Bottleneck Analysis

To address the limitations of current snapshot frameworks, a natural approach is to shift graphics resource uploads to the snapshot loading phase. However, in practice, we found that this method also introduces serious problems. When the upload is done asynchronously, where we initiates the upload of graphics resources but does not wait for completion before resuming VM execution, the GPU bandwidth remains heavily congested even after the VM resumes. This severely disrupts normal rendering, causing a sustained and significant FPS drop over an extended period. On the other hand, if the upload is done synchronously, which requires all resources to finish uploading before resuming execution, the snapshot load time increases dramatically to over 20 seconds for heavy-load apps. For app developers, such delays are intolerable, as frequent reloads during debugging would severely impact development efficiency. Applying uniform upload timing across all resources merely redistributes bottlenecks to different processing phases without fundamentally resolving latency issues. To address this, we explore feasible solutions through three representative case studies.

Case Study 1: Large but Infrequently Accessed Resources. Resources such as background textures and environment maps exemplify this category, characterized by their substantial size and low access frequency. Their large memory footprint results in a significant upload duration, but their infrequent use means they are often not needed immediately after the snapshot is loaded, with a high probability (95% in our measurements) that the upload will complete before the first access. This temporal gap between the start of the upload and the first access allows the upload process to complete in the background without disrupting normal rendering. By leveraging idle GPU bandwidth asynchronously, these resources can be uploaded efficiently while minimizing impact on the rendering pipeline. However, uploading all resources this way would overwhelm GPU bandwidth and harm rendering performance.

Case Study 2: Fragmented Small Resources. Resources such as UI elements exemplify this category, characterized by their small size but high access frequency. Their small memory footprint results in a minimal upload duration, but their frequent use means they are often needed immediately after the snapshot is loaded. When using lazy loading, these resources incur significantly higher overhead than larger ones, because each access triggers a synchronous upload, frequently interrupting the rendering pipeline. These interruptions often consume more time than the upload itself. Although asynchronous uploading appears more efficient, it still introduces extra overhead due to the need for synchronization primitives (e.g., `glWaitSync`) to guarantee that resources are fully uploaded before use, and this synchronization overhead can also surpass the upload time. In contrast, synchronous uploading avoids these runtime interruptions by uploading all resources upfront during the snapshot loading phase. Although this approach introduces a slight increase in initial loading time, it eliminates the need for costly sync operations during runtime, resulting in smoother performance.

Case Study 3: Short-Lived Resources. Resources such as particle effects, transient animations, or temporary UI overlays exemplify this category, characterized by their small size, short lifecycle, and large quantity. Their individual memory footprint is minimal, but their sheer number means that uploading them all at once would create a significant cumulative overhead. Lazy loading, however, distributes this overhead across multiple slack intervals, as these resources are accessed sporadically and briefly during their short lifecycle. This approach avoids the bottleneck of concentrated uploads while ensuring resources are available precisely when needed.

From these three representative cases, it is evident that each type of resource has its most suitable upload phase. Applying a one-size-fits-all approach fails to tailor the strategy to the unique characteristics of each resource. When each resource is allocated appropriately to avoid its specific bottleneck, latency can be reduced to acceptable levels in most cases rather than merely shifted. In summary, beyond the upload overhead itself,

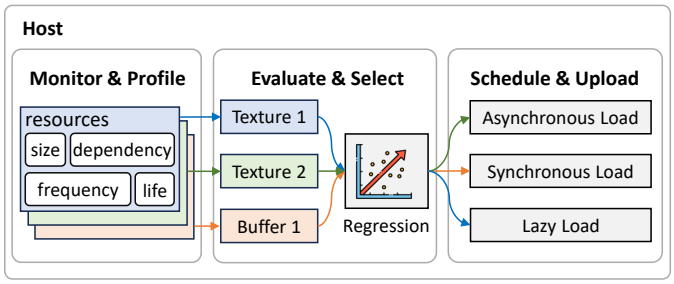


Figure 9: Adaptive VM snapshot design.

significant additional latency comes from context preparation, thread synchronization, and many other operations. Since their impact varies significantly by resource, these impacts collectively form unique bottleneck patterns that dominate different processing stages for each graphic resource. Moreover, while key constraints exist for resource uploads (*e.g.,* the snapshot loading time budget, limited frame intervals after resuming execution, and allocatable GPU bandwidth during uploads), we can maximize utilization of these finite resources by strategically scheduling uploads per resource type.

D. Adaptive Resource Uploading Strategy

We design and implement a snapshot loading approach that dynamically selects the optimal upload timing for each resource based on its unique characteristics. As shown in Figure 9, this process consists of three stages: monitoring resources, selecting strategies, and performing uploads. Below is a detailed explanation of each stage.

The newly introduced snapshot mechanism naturally preserves consistency across asynchronous devices without requiring any additional coordination. In both capture and loading phases, consistency is inherently guaranteed by design. During the capture phase, the system first completely pauses the guest CPU. Consequently, the host-side drivers also stop receiving new inputs and thus enter a static state. Although the data in our SVM framework maintain internal consistency constraints, these dynamic couplings no longer apply once the system is paused. As a result, both peripheral states and SVM data pipelines can be dumped into the snapshot file in any order without violating consistency.

For the loading phase, the situation is similar. The pipeline data structures within the SVM are loaded into memory before the system resumes execution. This allows them to immediately resume normal operation once the system restarts and naturally maintain device data coherence. Therefore, our design only needs to focus on efficiently restoring the resources of each individual device, most notably the GPU, without requiring additional mechanisms for device consistency.

Monitor and Profile Resource Characteristics. First, we must carefully consider the selection of specific resource characteristics. The most accessible features are static data (texture dimensions, memory footprint, format specifications, *etc.*), as they can be obtained during snapshot loading through specific APIs (e.g., `glGetTexParameter`). However, while static properties provide initial guidance, they cannot reveal dynamic behavioral patterns that critically affect upload strategy decisions. To overcome this limitation, we implement runtime instrumentation at key usage stages (*e.g.,* shader sampling, texture binding, and buffer mapping) to monitor and profile dynamic characteristics.

To optimize our model's selection effectiveness, we first conduct rigorous *metric selection* through comprehensive analysis of OpenGL ES resource behaviors. This process evaluates 17 candidate features spanning both static properties (*e.g.,* directly queryable attributes like texture dimensions via glGetTexParameteriv and memory footprint via glGetBufferParameteriv) and dynamic characteristics requiring instrumentation (*e.g.,* usage frequency patterns, dependency graphs, and lifecycle durations captured through runtime monitoring). We systematically eliminate theoretically insignificant factors (*e.g.,* texture format) and metrics with prohibitive collection overhead (*e.g.,* pixel-level access coherence requiring fragment shader hooks), ultimately identifying four decisive factors that satisfy our dual criteria of strategic relevance ($>$10% influence on upload latency) and measurement efficiency ($<$1% instrumentation overhead). These factors are *access frequency*, *resource dependency*, *lifecycle phase*, and *resource size*, which collectively provide robust insights into resource behavior while maintaining low measurement overhead.

To identify the strategic relevance values mentioned earlier, we conducted experiments on the hardware platform described in Section III-C. The core premise is that if a resource characteristic affects upload strategy selection, then the relative latency ratios between the three upload should change when the characteristic's value varies. Accordingly, we construct gradient test cases for each candidate metric. For example, we inject artificial resource access frequencies from 1 to 100 times per frame and vary resource lifetimes from single-frame to persistent. For these gradient test cases targeting specific characteristics, we use normalized variance to quantify their latency ratios across the three upload strategies, and calculate the variation range of this variance across different test cases as the strategic relevance value.

Evaluate and Select via Regression. We designed a lightweight scoring algorithm based on multiple linear regression to select the optimal upload strategy for each resource during snapshot loading. When storing snapshots, we record the characteristics of each resource through real-time monitoring (or direct acquisition). During snapshot loading, these characteristics are fed into the regression model to calculate scores for three strategies: synchronous upload, asynchronous upload, and lazy loading.

Specifically, our regression model maps resource features to scores by assigning a set of weight coefficients to each strategy. The score for each strategy is determined by a linear combination of access frequency, lifecycle phase, resource size, and resource dependency. To more accurately capture the nonlinear relationships between different features, we introduce feature interaction terms and piecewise linear regression. Feature interaction terms (e.g., the product of access frequency and resource size) can reflect the differences in upload overhead between large and small resources under high

access frequency, while piecewise linear regression assigns different weight coefficients based on resource size ranges (e.g., small, medium, and large resources), enabling more precise modeling of nonlinear feature effects.

The weights are initialized based on the quantitative results from our experimental data in the previous section. During snapshot loading, we record actual performance metrics and dynamically optimize the weight coefficients using the least squares method to minimize the error between predicted scores and actual performance. This optimization process updates the weights after each snapshot loading and stores them as new weight coefficients during the next snapshot save, ensuring continuous improvement of the model. In this way, the model gradually adapts to different app scenarios and system loads, improving the accuracy of strategy selection.

To further consider the competition among resources, we introduce global state variables (e.g., snapshot loading time budget, frame intervals, and GPU bandwidth utilization) and adjust the scores of each strategy using dynamic adjustment factors. When the number of asynchronously uploaded resources or GPU bandwidth utilization is high, the system reduces the score of asynchronous upload, thereby increasing the priority of synchronous upload. This mechanism ensures global optimization of resource allocation, avoiding performance degradation caused by excessive allocation of asynchronous upload resources, while dynamically adjusting the snapshot loading time budget and frame intervals to maintain a balanced proportion of the three strategies. For example, when GPU bandwidth utilization approaches its limit, the system tends to choose synchronous upload or lazy loading strategies to minimize interference with rendering performance.

Schedule and Perform Uploads. The resource upload implementation follows the strategy selected by our regression model, with several technical considerations. First, when handling dependent resources uploaded asynchronously or via lazy loading, we enforce upload ordering to ensure prerequisite resources are transferred before their dependents, thereby minimizing potential stalls. Second, for synchronous and asynchronous uploads, we optimize the process by batching multiple resources into consolidated transfers. This is implemented through a dedicated staging buffer that aggregates texture data before bulk submission to the GPU, significantly reducing per-resource overhead. This batch processing capability, which conventional lazy-loading frameworks cannot utilize, provides measurable efficiency improvements in our experiments.

VI. System Implementation

vSoC is based upon QEMU 7.1 [76], and hosts Android-x86 9.0 [77], as well as OpenHarmony 4.0 [78]. Since the implementation of vSoC only involves a set of guest-side drivers and a host-side QEMU device, vSoC can be easily ported to any higher version of QEMU, Android, and OpenHarmony. vSoC can run on Windows and macOS with Intel/AMD x86 CPUs. We choose the x86 platform because vSoC aims to run mobile systems on PCs and servers with heterogeneous hardware, which are primarily x86-based. The x86 choice is consistent with almost all mainstream mobile emulators. vSoC provides compatibility for ARM-based apps with the Intel Houdini binary translator [79].

In terms of implementation, the SVM framework includes a QEMU device (in VMX root mode) and a set of Linux guest kernel drivers (in VMX non-root mode). The virtual devices are built as components of the SVM framework in the host, and they are each accompanied with a guest kernel driver as well. Some virtual devices (*e.g.,* codec and GPU) are additionally accompanied by guest userspace drivers to integrate with the respective interfaces of mobile systems. Host-guest data transport in vSoC is based on the `virtio` [66] protocol, a de-facto standard used both in macrokernels like Linux and microkernels like seL4 [80].

To leverage the high-performance virtual GPU of the Trinity emulator [8], vSoC is built upon Trinity and we refactor its virtual GPU and its associated guest-host transport module with 55K and 4K lines of code changes. Since the virtual display from the guest's perspective is a window in the host, we implement it with `glfw` [81], a cross-platform library for window management. The virtual ISP (Image Signal Processor) utilizes Google Android Emulator's `YUVConverter` module [82] for in-GPU colorspace conversion of certain image formats, and `libswscale` [83], a software colorspace conversion library for other formats.

The adaptive snapshot loading approach is implemented based on QEMU's migration framework [76], which has seen wide adoption in cloud deployments and emerging secure migration scenarios, and provides a robust foundation for capturing and restoring the state of virtual devices. The majority of the implementation effort focuses on the GPU device, particularly the snapshot mechanism for graphics resources, which accounts for 10K lines of code. The remaining peripheral devices and transport pipes contribute an additional 500 lines. Notably, this implementation does not rely on any third-party libraries, ensuring a lightweight and self-contained solution.

VII. Evaluation

In this section, we answer the following questions about the performance of vSoC: (1) how does the SVM framework perform in practice (§VII-B), (2) how does vSoC perform with five types of heavy-load apps (§VII-C), (3) what is the contribution of individual designs (§VII-D), (4) effectiveness of the snapshot framework (§VII-E), and (5) how do top popular mobile apps receive the performance benefits (§VII-F).

A. Experimental Setup

Devices Under Test. To understand the performance of vSoC under heterogeneous hardware combinations, besides the high-end desktop PC used in the measurement (§III-C), we conduct our evaluation on a middle-end laptop PC as well. The component devices in the two machines are all prevalent commodity PC/server hardware: the high-end machine has a 24-core Intel i9-13900K CPU @ 3.0 GHz, 64 GB RAM (DDR5 4800 MHz), a NVIDIA RTX 3060 dedicated GPU, and a HIKVISION V148 USB camera; the middle-end machine has a 6-core Intel i7-10750H CPU @ 2.6 GHz, 16 GB RAM (DDR4 3200 MHz), a NVIDIA GTX 1660 Ti dedicated GPU,

Table III: SVM performance on the two PCs (high-end desktop vs. middle-end laptop).

Metric	vSoC	GAE	Q-K
Latency	0.34 / 0.38 *ms*	0.76 / 1.16 *ms*	0.22 / 0.25 *ms*
Coherence	2.38 / 3.45 *ms*	7.05 / 11.27 *ms*	6.15 / 9.28 *ms*
Throughput	3.49 / 3.24 GB/s	1.56 / 1.00 GB/s	0.96 / 0.89 GB/s

and an integrated webcam. Both machines run the emulators under Windows 11 23H2 version, except the Linux-exclusive QEMU-KVM, which we run on Ubuntu 22.04 LTS.

Target Emulators. Apart from vSoC, five mainstream mobile emulators are involved in the evaluation, including Google Android Emulator (GAE) [14], QEMU-KVM [15], LDPlayer [17], Bluestacks [16], and Trinity [8]. We configure every emulator instance with an 8-core CPU, 8 GB RAM, and a UHD (3820×2160) display. The display is configured with UHD resolution as opposed to Full-HD (1920×1080) to closely emulate heavy-load mobile platforms (*e.g.,* TVs and AR/VR headsets), which are usually equipped with high-resolution displays [84] including QHD, UHD, and even 8K.

Workloads. The apps involved are the 50 heavy-load apps listed in Table I, and the workloads are the same as those in §III-C. Each app is tested for 5 minutes on each emulator.

B. Microbenchmarks

Methodology. To characterize SVM performance, we measure its access latency, coherence time cost, and average throughput. SVM access latency and coherence cost are measured in the same way as that in §III-C by instrumenting the `AHardwareBuffer` interface and the emulators, while average throughput is calculated by dividing the total size of data accessed (excluding data wasted by broadcasting or prefetch failures) by the duration of the test. Since the prefetch protocol in vSoC makes multiple predictions, we additionally record the accuracy and the computation overhead of the predictions to ensure secure handling of prediction data. Since source code instrumentation is needed, only GAE and QEMU-KVM are involved for comparison.

Results. Table III shows the SVM performance of the three emulators. vSoC exhibits significantly lower coherence maintenance time cost than both GAE and QEMU-KVM (68% and 62% lower, respectively). That is because in vSoC, the majority (98%) of coherence maintenances are directly done in the host with the help of the SVM framework, eliminating the transport overhead across the virtualization boundary, and fully exploiting DMA capabilities of PC/server hardware. Since coherence maintenance time directly influences the throughput of a data pipeline (see §III-C), the average SVM throughput of vSoC is much higher than those of both GAE and QEMU-KVM (163% and 264% higher, respectively).

Due to the low-overhead prediction algorithm and various data caches, the CPU overhead of the various mechanisms and algorithms involved in vSoC (not including coherence maintenances) is kept to a negligible level (<1%). The maximum memory overhead of the various data structures of the SVM framework is 3.1 MiB.

C. Application Benchmarks

Methodology. The FPS and motion-to-photon latency of the heavy-load apps are measured. The two metrics are key indicators of the smoothness and responsiveness of mobile apps [85]. Incidentally, since no user input is involved during video playing, motion-to-photon latency is only measured on AR, camera, and livestream apps.

We collect FPS of an app using the `dumpsys` command from the Android Debug Bridge (ADB) shell [74]. For motion-to-photon latency, we use a high-speed camera to record videos of user interactions with the apps and examine each video to compute the latencies. The videos are recorded at a frame rate of 2000 FPS at Full-HD. For cloud/web-based livestreaming apps, we flash the screen contents of the emulators using the built-in developer tools of Android [86]. For the recorded videos, we examine each frame to recognize the timestamp when the user action happens and the timestamp when the corresponding response of the action manifests. In this way, the motion-to-photon latency can be calculated as the time delta of the two timestamps, with a negligible error of up to two frames (1.0 *ms*) introduced by the camera.

Results. Out of the 50 heavy-load apps, vSoC, GAE, QEMU-KVM, LDPlayer, Bluestacks, and Trinity can respectively run 48, 47, 42, 43, 44, and 20 of them. The criterion for a successful run is that the app does not report errors, crash, or produce Application-Not-Responding (ANR) [54] events during the 5-min test. Results of camera, AR, and livestream apps for Trinity are missing because Trinity does not support cameras or video encoders. The bar plots only contain performance data of apps that can be successfully run.

Figure 10 and Figure 11 show the FPS results of the application benchmarks. On the high-end machine, vSoC can achieve nearly full (57) FPS on all five types of heavy-load apps, with 82%, 160%, 292%, 656% and 797% better FPS on average than GAE, QEMU-KVM, LDPlayer, Bluestacks, and Trinity. We observe that while vSoC can run the heavy-load apps smoothly, other emulators all exhibit different levels of stuttering. Taking UHD video as an example, while GAE manages to play the videos at an acceptable framerate, videos often freeze for seconds on Bluestacks and LDPlayer. We also try to play lower-resolution videos (*e.g.,* 1280×720) on these emulators, and the results are smooth, indicating a performance problem rather than a functional one.

Interestingly, we observe that while Trinity performs very well in 3D gaming apps [8], it performs badly when running heavy-load apps. That is because Trinity is designed to minimize GPU virtualization overhead, and therefore works well on heavy-3D apps that extensively use GPU's rendering pipeline for real-time 3D rendering. vSoC does not outperform Trinity much on those heavy-3D apps. We evaluated vSoC using the same set of apps used in the Trinity evaluation. vSoC improves FPS of heavy-3D apps by only 1% on average—those apps rarely involve other SoC devices and shared memory. Trinity performs badly on heavy-load apps in the UHD/360 Video category, because Trinity only has a software virtual codec device inherited from Android-x86 [77]. Since Trinity's focus is on GPUs, it does not implement hardware

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2026.3667012

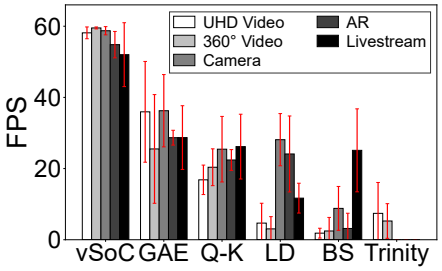


Figure 10: FPS on high-end PC.

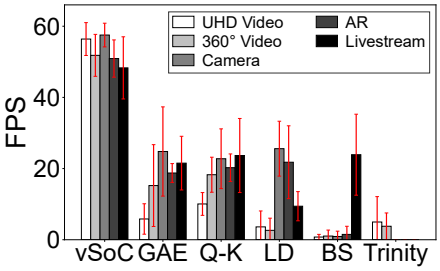


Figure 11: FPS on middle-end PC.

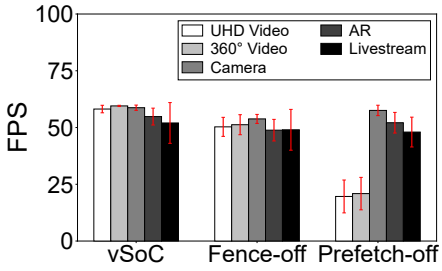


Figure 12: FPS breakdown on high-end PC.

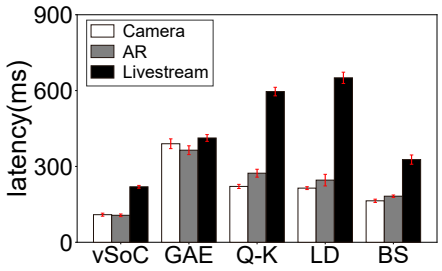


Figure 13: Latency on high-end PC.

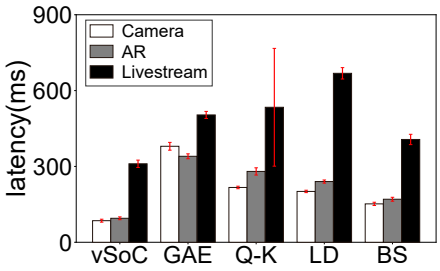


Figure 14: Latency on middle-end PC.

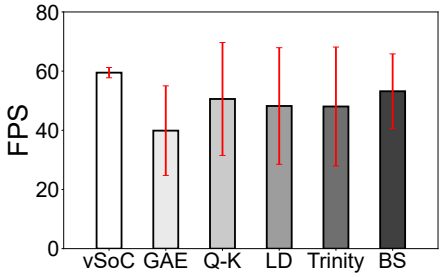


Figure 15: FPS of top apps on high-end PC.

codec devices or camera devices.

On the middle-end machine, where CPU and memory capabilities are relatively limited, vSoC manages to achieve an average FPS of 53, with 188%-1113% higher performance than the other emulators, as revealed in Figure 11. In particular, the performance of the video apps on GAE drops significantly (by 66%) on the middle-end PC. We observe that while video apps can perform at around 30 FPS on GAE at first, their performance quickly degrades to ˜10 FPS in one minute. After close inspection, we discover that the performance drop is due to CPU thermal throttling on the laptop PC, indicating further inefficiencies in the video decoder implementation of GAE.

Regarding the end-to-end latency of the apps, vSoC has the lowest latency among all the emulators, 62%, 60%, 61%, and 35% lower than GAE, QEMU-KVM, LDPlayer, and Bluestacks on the high-end machine (Figure 13). On the middle-end laptop, the end-to-end latency data are very similar to those on the high-end desktop machine (33%-61% lower, as shown in Figure 14), except that the latency of camera and AR apps are lower by 8 *ms* on average. That is because of the lower physical latency of the integrated camera on the laptop—we directly measure the camera latency on the two PCs using the `DirectShow` API [87] of Windows, and find out that the latency of the integrated camera is indeed 10 *ms* lower than the USB camera of the high-end desktop.

In summary, the study reveals that most users do perceive noticeable performance improvements in vSoC, especially for experienced users of heavy-load apps. Out of the 90 people crowdsourced, 83% perceive that vSoC is smoother than all the other emulators based on video playback smoothness perceived by eyes. 91% think that vSoC is less laggy than all the other emulators when using camera/AR apps, and 56% do not perceive any motion-to-photon latency in vSoC at all (the number is only 12% for the best of other emulators).

Overall, 43% of the participants believe that vSoC significantly improves their user experience compared to the other emulators, and the percentage rises to 93% for the 15 participants who identified themselves as frequent users of immersive apps. That aligns with existing studies which show that heavy-load apps like AR/VR require 60-180 FPS and sub-100 *ms* motion-to-photon latency to achieve a satisfying user experience and avoid motion sickness [1], [2].

D. Performance Breakdown

Methodology. To understand how individual components contribute to the performance of vSoC, we respectively disable the prefetch engine and the virtual fence mechanism, and repeat the evaluation on the high-end desktop machine. First, when the prefetch engine is disabled, we use the classic write-invalidate protocol [88] for coherence management instead. In the write-invalidate protocol, memory is lazily updated at the beginning of each SVM access (in the `begin_access` API of §III-A) to reduce memory bandwidth consumption.

Results. As shown in Figure 12, after the prefetch engine is turned off, the average performance of the heavy-load apps drops by 30%. In particular, the performance of the video apps drops by a staggering 66%. After a careful analysis of the video playback pipeline, we discover that the frame drop is caused by the high access latency of the write-invalidate protocol. To achieve real-time playback and avoid stale frames, the video renderer (`MediaCodec` [89]) of Android will assign a presentation timestamp to a video frame after it is decoded, indicating that GPU should finish rendering the video frame and present it to the user by the timestamp. Normally, video rendering is quick since it only involves sampling the video frame as a GPU texture, but unfortunately in emulation, when the GPU tries to access the video frame stored in the shared memory, coherence maintenance is triggered and blocks the

Table IV: Performance comparison of snapshot approaches

Approach	Stall Prob.	Stall Duration	Bugs Reproduced
GAE	94%	3284 ms	5%
vSoC-Lazy	83%	2761 ms	11%
Heuristic-Dep	47%	2563 ms	32%
Heuristic-Phase	42%	2434 ms	36%
Heuristic-Freq	39%	2317 ms	45%
Heuristic-Size	36%	2098 ms	49%
Heuristic-Avg	41%	2353 ms	41%
vSoC	19%	1816 ms	72%

render thread for up to 40.54 *ms*, causing many frames to miss the presentation deadline and get discarded.

Meanwhile, when the fence mechanism is disabled, we observe a moderate 11% decrease in FPS on all five categories of apps, showing that the virtual fence mechanism brings a more generic improvement on app performance. The extensive performance drop is reasonable since the mechanism is applied not only to the SVM framework, but to individual virtual devices like GPU as well (see §IV-D).

E. Snapshot Evalution

As snapshots are file-based and portable across emulators with identical configurations, with loading logic being independent of file availability, we conducted validation using two sets of snapshot files that maintain structural and contextual consistency. The test set consists of 100 pairs of heavy-load application snapshots from vSoC and GAE, where each pair maintain functional correspondence while permitting non-critical variances due to application dynamics.

Since GAE is inferior to vSoC in areas such as shared virtual memory management and generally has worse runtime performance, it is unfair to solely compare our adaptive snapshot with its snapshot. To provide a more balanced evaluation, we have added a breakdown-based comparison. In our snapshot framework, we disabled dynamic strategy selection and only used lazy loading, then compared it with the full version to ensure a more comprehensive analysis.

For post-load frame stalls, we define a stall event as any occurrence where FPS=0 persists for $>$32 ms (two consecutive 16 ms display cycles). The stall probability is calculated as the percentage of snapshots exhibiting frame stalls during the performance recovery phase, defined as the interval from load completion until the app achieves 90% of its post-warmup average FPS. Bug reproduction is verified through frame buffer comparison and exact call stack matching with original crash logs. The frame buffer comparison employs a pixel-level diffing algorithm that accounts for Android SurfaceFlinger-level variances, while the call stack is reconstructed through runtime instrumentation of the GPU driver's command processing pipeline to ensure state consistency.

As shown in Table IV, we conduct a comprehensive performance evaluation through three distinct comparisons: (1) against the original GAE system as baseline, (2) versus our framework's lazy-loading-only variant (vSoC-Lazy) for component-level analysis. Our experimental results demonstrate significant improvements: compared to GAE baseline, our multi-stage, adaptive snapshot framework reduces 75% post-load stalls and 45% per-stall duration, and improves bug reproduction rate by 14.4×. Notably, while the vSoC-Lazy variant achieves modest improvements over the GAE baseline, our full (vSoC) significantly outperforms vSoC-Lazy, highlighting the effectiveness of our adaptive algorithm.

To compare our regression model with simpler (i.e., more intuitive or straightforward) heuristics and demonstrate the necessity of adopting a learning-based approach, we conduct a specially designed set of experiments. We intentionally exclude overly complicated heuristics that mimic multi-feature decision processes because such rule-based designs are hard to generalize in practice. They can even add more design and implementation complexity than a learning-based regression model, which defeats the original purpose of using heuristics. Therefore, we design a set of baseline heuristics that reflect intuitive strategies based on access frequency, resource dependency, lifecycle phase, and resource size. These heuristics were carefully constructed to cover all three uploading strategie.

We evaluate them under the same experimental conditions, and the results are also presented in Table IV. As shown in our results, transitioning from naive lazy loading to a three-stage loading strategy brings substantial improvements in bug reproduction rate by 8.2×. This also explains why we regard stage-aware resource loading as a key scientific discovery. However, there remains a noticeable performance gap between the heuristics and the regression model, both in stability and effectiveness, which justifies the added effort of adopting a learning-based approach.

We further analyze the reasons behind this. First, during snapshot loading, the scheduling decision for each resource depends not only on its individual characteristics but also on the allocation of other resources and the overall system state. Under such constraints, simple heuristics are insufficient to find optimal allocations. In contrast, the regression model scores all resources in batches, jointly considering their features and system-level context to produce a globally optimal assignment in one pass. Second, the model is continuously updated after each loading round based on runtime feedback. As a result, it can swiftly adapt to diverse app scenarios and hardware configurations, which is a level of adaptability that fixed-rule heuristics fundamentally lack.

Additionally, we evaluate two sources of overhead introduced by our framework: the overhead introduced by the regression model and the cost of real-time graphics resource monitoring. The overhead of regression model mainly arises from two sources: model updates performed using feedback from the previous snapshot load before each save, and the inference latency during loading. We instrument the relevant code using `CLOCK_THREAD_CPUTIME_ID` to measure the overhead concurrently during the execution of the above experiments. On average, for heavy-load apps, model updates introduce a latency of 16 ms, while the total inference time for all resources during loading averaged 34 ms. Compared to the overall cost of snapshot saving and loading, both sources of overhead are negligible.

We then assess the impact of real-time graphics resource monitoring on performance during normal operation, supported by an additional set of experiments. This metric is reflected through runtime FPS measurements. Following the

same methodology used for measuring vSoC without monitoring, we conducted tests across two categories: 5 types of heavy-load applications and the top 25 most used applications. The results showed negligible impact, with merely 0.1% average FPS degradation observed in both test categories. This demonstrates that the monitoring overhead is virtually imperceptible in real-world usage scenarios.

Regression models are inherently subject to prediction uncertainty, and their potential operational consequences must be systematically evaluated. Theoretically, what we refer to as a "misprediction" is not an error in the strict sense, as there is no universally optimal stage for loading a given resource. Because the ideal decision point cannot be determined in advance, occasional deviations from the optimal point cannot be directly quantified through controlled experiments. Fortunately, any suboptimal predictions can still be inferred indirectly. When certain resources are not loaded at the most appropriate moments, the resulting overhead typically accumulates and manifests as frame stutters or performance drops after resumption. These effects are already captured by our post-resume performance metrics, which show that the current model achieves adequate predictive accuracy for most cases. Nonetheless, we acknowledge that imperfect classification still introduces some impact: our improvements increased the successful bug reproduction rate from 5% to 72%, but 28% of cases remain unaddressed, suggesting a promising direction for future refinement.

F. Impact on Top Popular Apps

Methodology. To further identify the impact of the design choices of vSoC on the performance of top popular apps besides the heavy-load ones, we conduct regression testing on the top-25 popular apps in Google Play, and record the FPS of the apps. The measurement methodology is the same as the application benchmarks in §VII-C. Moreover, we also carry out the performance breakdown experiment in §VII-D with the 25 popular apps.

Results. Of the top-25 popular mobile apps, vSoC, GAE, QEMU-KVM, LDPlayer, Bluestacks, and Trinity can respectively run 25, 21, 17, 25, 24, and 24 of them. As shown in Figure 15, vSoC respectively performs 49%, 18%, 23%, 24%, and 12% better than GAE, QEMU-KVM, LDPlayer, Bluestacks, and Trinity in terms of average FPS. The bar plots only contain performance data of apps that can be successfully run. The results are very close to the bar plots as well: vSoC achieves 12%-49% better FPS. vSoC can achieve moderate performance improvements when running popular apps as well, because SVM is also commonly used in other system components of the Android framework (*e.g.,* Skia [90]) besides specialized data pipelines, and therefore the SVM improvements of vSoC can benefit them as well.

Regarding performance regression, vSoC performs slightly worse than the best of other emulators on 5 popular apps, with 4% fewer FPS on average. In fact, the 5 apps run at nearly 60 FPS (avg. 57.4 FPS) on vSoC. The reason behind the negligible performance regression is that the SVM framework in vSoC is on-demand and brings little overhead when the app/system does not use it. Regarding performance breakdown, when the prefetch engine or the fence mechanism is disabled, 20 (80%) and 24 (96%) apps experience frame rate drops respectively, and the average FPS of the apps decreases respectively by 6% and 8%, indicating that the design of vSoC can also moderately improve the FPS of ordinary apps. Of the apps that do not experience frame rate drops, we discover that they all run at ˜60 FPS regardless of whether the mechanisms are enabled, indicating that their workloads are relatively simple.

VIII. Conclusion

This paper presents vSoC, the first virtual mobile SoC that enables efficient emulation of high-throughput SoC devices. The key methodology of vSoC is to break away from the traditional modular architecture of virtual devices by establishing a unified framework for shared virtual memory. Our evaluation demonstrates that vSoC achieves significant performance improvements for heavy-load applications that heavily rely on a variety of SoC devices. This performance gain has led to the adoption of vSoC by a major mobile app IDE, showcasing its practical impact.

To meet the debugging requirements posed by the developers of heavy-load apps, we further develop an adaptive snapshot restoration approach that effectively addresses the critical post-load stalling issues exhibited in existing mobile emulators. Our approach provides reliable bug reproduction capabilities for heavy-load apps with little overhead.

We hope that the experiences involved in this work can shed light on the usage of shared memory in mobile systems, contribute to the emulation of other platforms with hardware disparities, and open up opportunities for future use cases of mobile systems such as cloud-based AR/VR.

Acknowledgments

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.

References

[1] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai, "Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices," in *Proc. of ACM MobiCom*, 2017, pp. 409–421.

[2] W. Zhang, B. Han, and P. Hui, "SEAR: Scaling Experiences in Multi-User Augmented Reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, pp. 1982–1992, 2022.

[3] LG Electronics, "What Is 4K TV Resolution & Why It's The Best," 2024, https://www.lg.com/ae/lg-story/helpful-guide/what-is-4k-resolution.

[4] AUTOBOOM, "Arcfox Alpha T - Generations, Types of Execution and Years of Manufacture," 2024, https://autoboom.co.il/en/catalog/cars/arcfox/alpha-t.

[5] A. D'Antras, C. Gorgovan, J. Garside, and M. Luján, "Low Overhead Dynamic Binary Translation on ARM," in *Proc. of ACM PLDI*, 2017, pp. 333–346.

[6] M. Dowty and J. Sugerman, "GPU Virtualization on VMware's Hosted I/O Architecture," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 73–82, 2009.

[7] Q. Yang, Z. Li, Y. Liu, H. Long, Y. Huang, J. He, T. Xu, and E. Zhai, "Mobile Gaming on Personal Computers with Direct Android Emulation," in *Proc. of ACM MobiCom*, 2019, pp. 1–15.

[8] D. Gao, H. Lin, Z. Li, C. Huang, Y. Liu, F. Qian, L. Gong, and T. Xu, "Trinity: High-Performance Mobile Emulation through Graphics Projection," in *Proc. of USENIX OSDI*, 2022, pp. 285–301.

[9] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "AvA: Accelerated Virtualization of Accelerators," in *Proc. of ACM ASPLOS*, 2020, pp. 807–825.

[10] H. Raj and K. Schwan, "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices," in *Proc. of ACM HPDC*, 2007, pp. 179–188.

[11] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat, "PicNIC: Predictable Virtualized NIC," in *Proc. of ACM SIGCOMM*, 2019, pp. 351–366.

[12] R. A. Baratto, S. Potter, G. Su, and J. Nieh, "MobiDesk: Mobile Virtual Desktop Computing," in *Proc. of ACM MobiCom*, 2004, pp. 1–15.

[13] W. Chen, L. Xu, G. Li, and Y. Xiang, "A Lightweight Virtualization Solution for Android Devices," *IEEE Transactions on Computers*, vol. 64, pp. 2741–2751, 2015.

[14] Google, "Google Android Emulator," 2024, https://developer.android.com/studio/run/emulator.

[15] QEMU Developers, "QEMU Official Website," 2024, https://www.qemu.org/.

[16] now.gg, "BlueStacks Gaming Platform for PC," 2024, https://www.bluestacks.com/.

[17] ldplayer.net, "LDPlayer Android Emulator for PC," 2024, https://www.ldplayer.net.

[18] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, 1989.

[19] Google, "The VSync Mechanism," 2024, https://source.android.com/docs/core/graphics/implement-vsync.

[20] E. S. Gardner Jr., "Exponential Smoothing: The State of the Art," *Journal of Forecasting*, vol. 4, pp. 1–28, 1985.

[21] J. Qiu, Z. Zhou, Y. Li, Z. Li, F. Qian, H. Lin, D. Gao, H. Su, X. Miao, Y. Liu, and T. Xu, "vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware," in *Proc. of ACM SOSP*, 2024, pp. 558–573.

[22] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," in *Proc. of ACM PPoPP*, 1990, pp. 168–176.

[23] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proc. of ICPP*, 1988.

[24] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking Software Runtimes for Disaggregated Memory," in *Proc. of ACM ASPLOS*, 2021, pp. 79–92.

[25] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, 1994, p. 10.

[26] T. Allen and R. Ge, "In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing," in *Proc. of ACM/IEEE SC*, 2021, pp. 1–15.

[27] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory," in *Proc. of ACM/IEEE ISCA*, 2019, pp. 224–235.

[28] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs," *Proceedings of the VLDB Endowment*, vol. 14, pp. 114–127, 2020.

[29] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proc. of ACM ASPLOS*, 1996, pp. 174–185.

[30] B. Suchy, S. Campanoni, N. Hardavellas, and P. Dinda, "CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation," in *Proc. of ACM PLDI*, 2020, pp. 329–345.

[31] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator," *IEEE Concurrency*, vol. 8, pp. 12–20, 2000.

[32] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott, "VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks," *ACM SIGARCH Computer Architecture News*, vol. 25, pp. 157–169, 1997.

[33] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs," *ACM Transactions on Architecture and Code Optimization*, vol. 13, pp. 35:1–35:25, 2016.

[34] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed Shared Memory with In-Network Cache Coherence," in *Proc. of USENIX FAST*, 2021, pp. 277–292.

[35] A. Kayi, O. Serres, and T. El-Ghazawi, "Adaptive Cache Coherence Mechanisms with Producer–Consumer Sharing Optimization for Chip Multiprocessors," *IEEE Transactions on Computers*, vol. 64, pp. 316–328, 2015.

[36] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-Based Near-Memory Processing with Inter-DIMM Broadcast," in *Proc. of ACM/IEEE ISCA*, 2021, pp. 237–250.

[37] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives," in *Proc. of ACM/IEEE ISCA*, 2020, pp. 996–1009.

[38] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 40–52, 1991.

[39] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel Virtualization Technology," *Computer*, vol. 38, pp. 48–56, 2005.

[40] C. Clark and K. Fraser, "Live Migration of Virtual Machines," in *Proc. of USENIX NSDI*, 2005.

[41] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-Copy Live Migration of Virtual Machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 14–26, 2009.

[42] H. Liu, C.-Z. Xu, H. Jin, J. Gong, and X. Liao, "Performance and Energy Modeling for Live Migration of Virtual Machines," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, 2011, pp. 171–182.

[43] A. Strunk, "Costs of Virtual Machine Live Migration: A Survey," in *2012 IEEE Eighth World Congress on Services*, 2012, pp. 323–329.

[44] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation," in *Cloud Computing*, 2009, pp. 254–265.

[45] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive Process-Level Live Migration in HPC Environments," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[46] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud," in *Proc. of ACM SoCC*, 2022, pp. 78–93.

[47] P. Wang, Y. Li, C. Fang, Y. Zhong, and Z. Ding, "Optimizing Serverless Performance Through Game Theory and Efficient Resource Scheduling," *IEEE Transactions on Computers*, vol. 74, pp. 1990–2002, 2025.

[48] Q. Liu, Y. Yang, D. Du, Y. Xia, P. Zhang, J. Feng, J. R. Larus, and H. Chen, "Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu," in *Proc. of USENIX ATC*, 2024, pp. 1–17.

[49] Q. Zeng, K. Hou, X. Leng, and Y. Chen, "DirectFaaS: A Clean-Slate Network Architecture for Efficient Serverless Chain Communications," in *Proc. of ACM WWW*, 2024, pp. 2759–2767.

[50] M. Dowty and J. Sugerman, "GPU Virtualization on VMware's Hosted I/O Architecture," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 73–82, 2009.

[51] A. M. Lai and J. Nieh, "On the Performance of Wide-Area Thin-Client Computing," *ACM Trans. Comput. Syst.*, vol. 24, pp. 175–209, 2006.

[52] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, vol. 2, pp. 33–38, 1998.

[53] M. PratapSingh and M. Kumar Jain, "Evolution of Processor Architecture in Mobile Phones," *International Journal of Computer Applications*, vol. 90, pp. 34–39, 2014.

[54] Google, "Android Application-Not-Responding," 2024, https://developer.android.com/topic/performance/vitals/anr.

[55] OpenHarmony, "OpenHarmony OH_NativeBuffer Interface," 2024, https://gitee.com/openharmony/graphic_graphic_surface.

[56] Google, "Hardware Abstraction Layer Overview," 2024, https://source.android.com/docs/core/architecture/hal.

[57] Khronos, "OpenGL ES Official Website," 2011, https://www.khronos.org/opengles/.

[58] ——, "OpenMAX Official Website," 2011, https://www.khronos.org/openmax/.

[59] Google, "Android Camera HAL," 2024, https://source.android.com/docs/core/camera/camera3.

[60] J. Ajanovic, "PCI Express 3.0 Overview," in *Proc. of IEEE HCS*, 2009, pp. 1–61.

[61] Linux Developers, "Linux Kernel Memory Allocation Guide," 2024, https://www.kernel.org/doc/html/v6.0/core-api/memory-allocation.html.

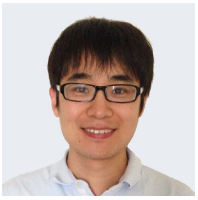
[62] ——, "Linux Manual Page of Mmap," 2023, https://man7.org/linux/man-pages/man2/mmap.2.html.

[63] Google, "ARCore Official Website," 2024, https://developers.google.com/ar.
[64] Adobe Systems, "Adobe RTMP Specification," 2019, https://rtmp.veriskope.com/docs/spec/.
[65] Will Reese, "Nginx: The High-Performance Web Server and Reverse Proxy," 2008, https://dl.acm.org/doi/fullHtml/10.5555/1412202.1412204.
[66] Kernel Developers, "Virtio on Linux — The Linux Kernel Documentation," 2024, https://docs.kernel.org/driver-api/virtio/virtio.html.
[67] Google, "Android AHardwareBuffer Interface," 2024, https://developer.android.com/ndk/reference/struct/a-hardware-buffer-desc.
[68] H. Lin, Z. Li, D. Gao, Y. Liu, F. Qian, T. Xu, B. Xiao, and X. Qin, "Trinity: High-Performance and Reliable Mobile Emulation through Graphics Projection," *ACM Trans. Comput. Syst.*, vol. 42, pp. 6:1–6:33, 2024.
[69] S. J. Taylor and B. Letham, "Forecasting at Scale," *The American Statistician*, vol. 72, pp. 37–45, 2018.
[70] Alibaba, "Elastic Desktop Service (EDS) - Virtual DesktopAlibaba Cloud," https://www.alibabacloud.com/product/cloud-desktop.
[71] Amazon, "Amazon WorkSpaces Client Download," https://clients.amazonworkspaces.com/.
[72] P. Maiya and A. Kanade, "Efficient Computation of Happens-before Relation for Event-Driven Programs," in *Proc. of ACM ISSTA*, 2017, pp. 102–112.
[73] B. Hailpern and P. Santhanam, "Software Debugging, Testing, and Verification," *IBM Systems Journal*, vol. 41, pp. 4–12, 2002.
[74] Google, "Android Debug Bridge," 2024, https://developer.android.com/tools/adb.
[75] ——, "Logcat Command-Line Tool," 2025, https://developer.android.com/tools/logcat.
[76] QEMU Developers, "QEMU Version 7.1.0," 2022, https://www.qemu.org/2022/08/30/qemu-7-1-0/.
[77] Chi-Wei Huang, "Android-X86 Release 9.0-R2," 2024, https://www.android-x86.org/releases/releasenote-9-0-r2.html.
[78] OpenHarmony, "OpenHarmony Official Website," 2024, https://gitee.com/openharmony.
[79] Intel.com, "Houdini: Translate The ARM Binary Code into the X86 Instruction Set," 2021, https://www.intel.com/content/www/us/en/products/systems-devices/workstations.html.
[80] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proc. of ACM SOSP*, 2009, pp. 207–220.
[81] GLFW Developers, "GLFW: An OpenGL Library," 2024, https://www.glfw.org/.
[82] Android Open Source Project, "YuvConverter.Java Android Open Source Project," 2024, https://chromium.googlesource.com/external/webrtc/+/HEAD/sdk/android/api/org/webrtc/YuvConverter.java.
[83] Libswscale Developers, "Libswscale Documentation," 2024, https://ffmpeg.org/libswscale.html.
[84] M. Nebeling, S. Rajaram, L. Wu, Y. Cheng, and J. Herskovitz, "XRStudio: A Virtual Production and Live Streaming System for Immersive Instructional Experiences," in *Proc. of ACM CHI*, 2021, pp. 1–12.
[85] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld, "An Evaluation of QoE in Cloud Gaming Based on Subjective Tests," in *Proc. of Springer IMIS*, 2011, pp. 330–335.
[86] Google, "Android On-Device Developer Options," 2024, https://developer.android.com/studio/debug/dev-options.
[87] stevewhims, "DirectShow - Win32 Apps," 2023, https://learn.microsoft.com/en-us/windows/win32/directshow/directshow.
[88] N. P. Jouppi, "Cache Write Policies and Performance," *ACM SIGARCH Computer Architecture News*, vol. 21, pp. 191–201, 1993.
[89] Google, "Android MediaCodec," 2024, https://developer.android.com/reference/android/media/MediaCodec.
[90] ——, "Skia Official Website," 2024, https://skia.org/.

Zijie Zhou (Student Member, IEEE) received the B.Sc degree in software engineering from the School of Software, Nanjing University in 2023. She is working towards the Ph.D. degree at the School of Software, Tsinghua University. Her research areas mainly include mobile operating system and emulator.

Jiaxing Qiu (Student Member, IEEE) received the BA degree in English from Department of Foreign Languages and Literatures, Tsinghua University in 2022. He is working towards the Ph.D. degree at the School of Software, Tsinghua University. His research areas mainly include mobile systems and virtualization.

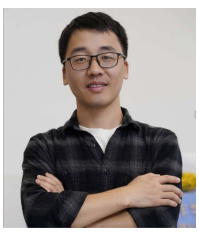
Zhenhua Li (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees from Nanjing University, in 2005 and 2008 respectively, and the Ph.D. degree from Peking University in 2013, all in computer science and technology. He is a Tenured Associate Professor with the School of Software, Tsinghua University. His research areas cover mobile networking/emulation and cloud computing/rendering.



Feng Qian (Senior Member, IEEE) received the B.Sc. degree from Shanghai Jiao Tong University and the Ph.D. degree from the University of Michigan. He is an Associate Professor in the Ming Hsieh Department of Electrical and Computer Engineering, Viterbi School of Engineering at University of Southern California. His research interests cover mobile networking, AR/VR, wearable computing, and system security.

Yunhao Liu (Fellow, IEEE) received the B.Sc. degree from the Automation Department at Tsinghua University, an M.Sc. degree and a Ph.D. degree in computer science and engineering from Michigan State University. He is a Full Professor and the Dean of Global Innovation Exchange (GIX), Tsinghua University. His research interests include sensor networking, IoT, RFID, and cloud computing.

Bo Xiao Technical Expert at Ant Group, received the B.Sc. from Huaqiao University and an M.Sc. from the University of Electronic Science and Technology of China. Currently based in the Multimedia Technology Team at Ant Group (Chengdu, China), his research focuses on cloud rendering and graphics/image processing systems.



Hongwei Hu Senior Technical Expert at Ant Group, currently overseeing multimedia technology and multimodal interaction technologies. Previously led audio and video product technology teams at Alibaba Group, DingTalk, and Alibaba Cloud. Primary research and work domains include video processing, real-time communication (RTC), codecs, cloud rendering.