vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware

Jiaxing Qiu[1], Zijie Zhou[1], Yang Li[1], Zhenhua Li[1]✉, Feng Qian[2]
Hao Lin[1,3], Di Gao[1], Haitao Su[1], Xin Miao[1], Yunhao Liu[1], Tianyin Xu[3]
[1]Tsinghua University [2]University of Southern California [3]University of Illinois Urbana-Champaign

Abstract

Emerging mobile apps such as UHD video and AR/VR access diverse *high-throughput* hardware devices, e.g., video codecs, cameras, and image processors. 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 and host's memory architectures for hardware devices, *i.e.,* the mobile guest's centralized memory on a system-on-chip (SoC) versus the PC/server's separated memory modules on individual hardware. Such a discrepancy makes the shared virtual memory (SVM) architecture of mobile emulators highly inefficient.

To address this, we design and implement vSoC, the first virtual mobile SoC that enables virtual devices to efficiently share data through a *unified* SVM framework. We then build upon the SVM framework a prefetch engine that effectively hides the overhead of coherence maintenance (which guarantees that devices sharing the same virtual memory see the same data), a performance bottleneck in existing emulators. Compared to state-of-the-art emulators, vSoC brings 12%-49% higher frame rates to top popular mobile apps, while achieving 1.8-9.0× frame rates and 35%-62% lower motion-to-photon latency for emerging apps. vSoC is adopted by Huawei DevEco Studio, a major mobile IDE.

CCS Concepts: • **Software and its engineering → Virtual machines**; • **Computer systems organization → System on a chip**.

Keywords: virtualization, mobile systems, system-on-chip, shared memory.

ACM Reference Format:
Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, Tianyin Xu. 2024. vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3694715.3695946

1 Introduction

Mobile emulation enables mobile OSes and apps to execute seamlessly on PC/server machines. Recently, mobile OSes and apps have been widely deployed on diverse platforms such as televisions, smart cars, and AR/VR headsets. Different from traditional mobile apps, emerging apps on such platforms (e.g., UHD video streaming and AR/VR) need much better performance (4-16× resolutions, 60-180 FPS, and sub-100ms motion-to-photon latency) to deliver a satisfying user experience [6, 47, 49, 86]. Moreover, they intensively interact with a variety of high-throughput System-on-Chip (SoC) devices such as video codec, image signal processor (ISP), and 2D/3D camera. Consequently, app developers have even stronger demands for mobile emulators that can emulate the entire mobile SoC efficiently, rather than individual devices in previous work [7, 12, 15, 16, 18, 46, 58, 74, 84, 85].

Unfortunately, state-of-the-art emulators [18, 28, 48, 65, 73] (including Trinity [18], a high-performance mobile emulator that can smoothly run GPU-intensive apps) exhibit poor performance when running the emerging apps. They constantly suffer from performance issues such as video stalls and high motion-to-photon latency, compared to executions on real mobile devices. With in-depth instrumentation of the emulators, we pinpoint the major reason to be *inefficient data sharing among virtual 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, where a single physical memory is connected to CPU and other devices. In contrast, many PC/server devices have dedicated local memory; they exchange data with the main memory via buses. Given this hardware architecture discrepancy, emulators have to adopt a shared virtual memory (SVM) architecture [51]. SVM presents an illusion of a unified address space to the guest mobile OSes from physically distributed host memory. This is achieved by transparently maintaining the *data coherence* among devices and the main memory, *i.e.,* devices sharing the same virtual memory should see the same data.

Efficient SVM coherence maintenance is a key challenge to virtualizing an SoC. Heavily inspired by the modular nature

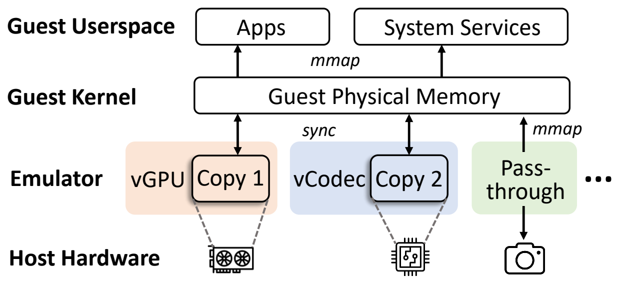


Figure 1. Memory architecture of typical emulators.

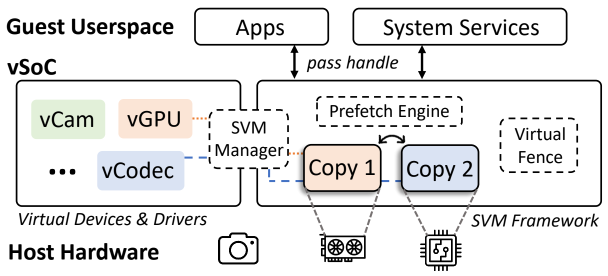


Figure 2. Memory architecture of vSoC.

of PC/server virtualization, virtual devices in existing mobile emulators [28, 73] are largely agnostic of each other. As a result, coherence is usually maintained with the help of the guest (as demonstrated in Figure 1): a piece of guest memory is mapped to guest system services and apps, and each virtual device only needs to synchronize its local copy with the guest memory and keep it up to date. Nevertheless, when shared memory is used to move data between devices, this architecture can lead to a waste of memory bandwidth since data have to be frequently copied to and from the guest.

Worse still, 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. For example, the write-invalidate protocol [36] brings high SVM access latency in practice—when data size is large, coherence maintenance can block the next SVM access for milliseconds. The blocking is oftentimes unexpected to the mobile OS, eventually causing response delays of hundreds of milliseconds and visible frame drops in the apps.

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. An immediate advantage of vSoC is that it 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 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 ensure this, today's OSes employ various mechanisms such as buffering and VSync [32], which lead to unavoidable delays (avg. 17 *ms* based on our measurement) between consecutive shared memory accesses. We refer to such delays as *slack intervals*.

While slack intervals are unavoidable, vSoC makes use of them through *prefetching*: it predicts when and where the next SVM access would occur, and fetches data updates to the predicted device ahead of time. In this way, coherence maintenance is hidden under slack intervals, thus boosting the SVM performance. While prefetching has been widely used in computer systems, designing a prefetch engine for vSoC faces three unique challenges: how to robustly predict SVM accesses; how to accommodate prefetching to different slack interval durations; and how to guarantee the ordering of SVM operations among host devices. We next highlight our solutions to these challenges.

To foster robust SVM access prediction, vSoC leverages its global vantage point to collect various SVM usage data, properly maintained as two-layer *hypergraphs*, and uses them to 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 [19]) can achieve an SVM access prediction accuracy of ≥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.

The cross-device nature of prefetch brings the third challenge for efficient ordering of SVM commands among host devices. The key problem is that command orders are only known to guest drivers, but need to be enforced in the host, leading to frequent guest-host control flow synchronizations with high overhead [18, 81]. To solve the problem, we attach order semantics to the commands dispatched by the guest with *virtualized* fence instructions. The virtual fences take effect in pairs and represent *happens-before* relationships [61], which are enforced in the host by translating the virtual fences to device-specific locks and memory fences.

We integrate the above designs and implement vSoC for Android and OpenHarmony, two Linux-based mobile systems, in 91K lines of C/C++ code. Note that vSoC is built atop Trinity [18] to inherit its high-performance virtual GPU. We compare vSoC's performance against five mainstream mobile emulators (Google Android Emulator [28], QEMU-KVM [73], LDPlayer [48], Bluestacks [65], and Trinity [18]) with SVM microbenchmarks, top-25 popular mobile apps, and 50 emerging apps (that extensively use SoC devices) including UHD video, 360-degree video, AR/VR, camera, and livestream. Compared to other emulators, vSoC brings 12%-49% higher FPS to the top popular apps, while achieving 1.8-9.0× FPS and 35%-62% lower motion-to-photon latency for the emerging apps. vSoC is adopted by Huawei DevEco Studio [33], a major commercial mobile IDE.

In summary, the paper makes the following contributions:

- We present vSoC, the first virtual mobile SoC that breaks free from the classical modular architecture of virtual devices for ahead-of-time coherence management.
- We design a low-overhead access ordering mechanism for vSoC with virtual memory fences, efficiently supporting shared resource operations across virtual devices.
- We implement vSoC for two open-source mobile systems and show that it can achieve considerable performance improvements on a variety of applications.

The code and data of vSoC are publicly available at https://github.com/virtualsoc/vsoc.

2 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 (§2.1), how the abstraction is implemented in typical emulators (§2.2), reveal the real-world characteristics of shared memory (§2.3), and finally, discuss its implications for the design of vSoC (§2.4).

2.1 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 [70], 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 [21] in Android, and the `OH_NativeBuffer` interface [67] in OpenHarmony. The interface sits at the Hardware Abstraction Layer (HAL) [29] of a mobile system, which is typically called by system services or apps and implemented by SoC manufacturers along with other device drivers.

```
// allocates a shared memory region and returns a
// handle pointing to the region.
int alloc(struct shmem_module_t* module,
        struct region_t size,
        buffer_handle_t* handle);

// frees a shared memory region.
int free(struct shmem_module_t* module,
        buffer_handle_t handle);

// begins an access to the shared memory.
// 'usage' specifies if the access is RO, WO or RW.
// only the region specified by 'size' will be accessed.
// the virtual address is returned in 'vaddr'.
int begin_access(struct shmem_module_t const* module,
        buffer_handle_t handle, struct region_t size,
        int usage, void** vaddr);

// ends the access to the shared memory.
int end_access(struct shmem_module_t const* module,
        buffer_handle_t handle);
```

Figure 3. The shared memory interface of mobile systems.

A shared memory interface typically consists of the APIs shown in Figure 3, 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 [41], OpenMAX [42], Camera HAL [22]), so the handles can be directly passed to other SoC devices. In this way, data can be shared among mobile SoC devices efficiently.

2.2 Shared Virtual Memory in Mobile Emulation

While a mobile SoC achieves high power and space efficiency through a monolithic architecture, PC/server devices are usually modular: they are connected to the main memory via buses like PCI-e [2], 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 [51], 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. For instance, suppose two virtual devices (V_a and V_b) use two different physical devices (P_a and P_b) in the host. The guest OS first commands V_a's driver to write to an SVM region and then commands V_b's driver to read from it. In the host, P_a creates its copy of the SVM region and writes to it, but P_b cannot access P_a's device memory. Therefore, the emulator needs to copy the data from P_a to P_b before P_b accesses the data—the data copying is termed *coherence maintenance*.

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 [28] 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` [57] and mapped to userspace via `mmap` [56], 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 emerging mobile apps, as will be revealed below.

2.3 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. Since OpenHarmony is a recent mobile system and does not have a mature app ecosystem yet [68], OpenHarmony is not included in the measurement.

Workloads. As shown in Table 1, we choose 50 emerging 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 (as of Mar. 2024). For AR apps, since the Google ARCore [27] 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.

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

Table 1. The five types of emerging 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

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 [1], 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 [83], 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) [28] and QEMU-KVM [73]. 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 (§2.1) 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 [65] in the measurement.

Observations. Shared memory is frequently used in all the emerging 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).

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

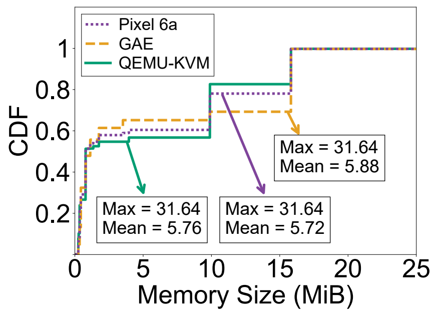


Figure 4. Size of shared memory on the three platforms.

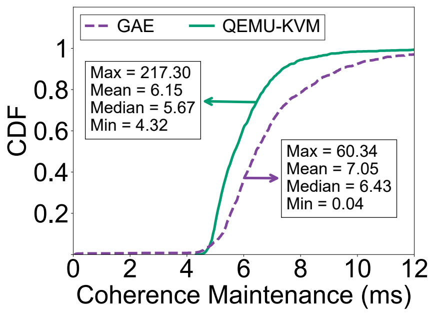


Figure 5. Coherence time cost of the two emulators.

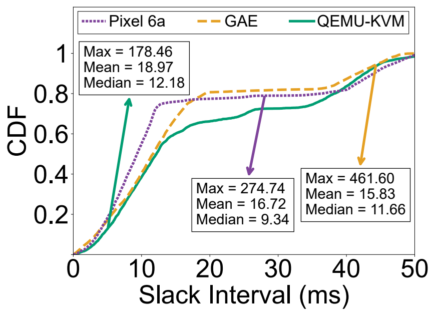


Figure 6. Slack intervals of the three platforms.

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 inter-process communications (IPC) as well.

Moreover, as shown in Figure 4, 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 5, 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 6, 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 [32]) 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 6, 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.

2.4 Implications for vSoC

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

First, slow coherence maintenance is a key performance problem in existing emulators. Each data pipeline involves one or more coherence maintenances, which each takes >6 *ms* to complete, because of the inefficient data copies across the virtualization boundary (see §2.2). As a reference, for an app to run smoothly at 60 FPS, only 16.7 *ms* is allowed for a video frame to go through a data pipeline. Slow coherence maintenances occupy the already tight time budget for app frames, leave less room for the actual data processing, and ultimately lead to slow and dropped frames.

Another key implication is that we can make use of the slack intervals in mobile systems to boost SVM performance. The idea is to use a prefetch coherence protocol: it tracks the data flows associated with SVM regions, predicts the next accessing device according to historical usage, and fetches data updates to the predicted device during the slack intervals. In this way, coherence maintenance is hidden under slack intervals, significantly reducing the time overhead.

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 (§2.2). 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 time and bandwidth overhead, 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.

3 System Design

This section presents the internals of vSoC. §3.1 covers the overall architecture of vSoC, while §3.2, §3.3 and §3.4 detail the design choices made in various components of vSoC.

3.1 Overview

vSoC sets out to meet the following design goals: (1) providing a unified virtualization of shared memory for the virtual devices; (2) designing an efficient coherence protocol in the context of mobile emulation; (3) addressing other performance issues of existing SVM architectures.

To achieve the above design goals, vSoC breaks through the boundary of virtual devices. As illustrated in Figure 2, vSoC features an SVM framework that provides an efficient SVM architecture for all the virtual devices. The SVM framework itself is paravirtualized, including a host virtual device and a guest kernel driver based on `virtio` [40] (the kernel itself is unchanged). We choose paravirtualization instead of other I/O virtualization techniques, because during app development, hardware resources typically need to be shared between the guest and the host. Nevertheless, since vSoC primarily solves a memory architecture problem, its high-level design is applicable to other virtualization techniques like PCIe pass-through and containers as well.

The SVM framework consists of three components: SVM Manager (§3.2), responsible for providing a unified internal representation of SVM; Prefetch Engine (§3.3), that realizes a robust prefetch coherence protocol for SVM; and Virtual Command Fence (§3.4), which provides a low-overhead access ordering mechanism for host operations. Powered by the SVM framework, vSoC also includes a common set of paravirtualized SoC devices, including GPU, display, ISP, codec, camera, and cellular modem, which each has its own host-side module and guest kernel driver.

3.2 The SVM Manager

The SVM Manager implements the shared memory interface (§2.1) of mobile systems and manages the lifecycle of SVM resources for vSoC.

Every SVM region is assigned a unique 64-bit ID upon allocation. The memory space of each SVM region is lazily allocated, because the actual device accessing the SVM region can only be known when the first access occurs. Since only the host emulator should be permitted to access host memory to ensure guest-host isolation, most SVM data operations must be done by the host. Therefore, regarding each SVM region, the guest only caches a portion of metadata (*e.g.,* size) to allow quick responses to control operations of SVM. The complete metadata and actual resource handles (*e.g.,* a scatterlist of guest memory, or a handle to GPU memory) required to perform actual data operations are maintained in a host-side hashtable.

Owing to the unified representation offered by the SVM Manager, virtual devices (and drivers) of vSoC can use the unique ID to identify an SVM region, instead of carrying the actual data with the device commands and worrying about coherence maintenances. An immediate benefit of the unified architecture is that, when shared memory is used to transfer data between devices, coherence can be directly maintained among virtual devices without guest involvement, instead of having to rely on guest memory and result in extra bandwidth consumption (§2.2).

Taking camera streaming as an example, the camera writes to an SVM region which is read by the GPU afterwards. With the old architecture (in §2.2), the process will trigger four memory copies: (1) The image in the SVM is copied from the camera hardware to the host memory. (2) The camera virtual device copies the image in the host memory to the guest memory corresponding to the SVM region.[3] (3) The GPU virtual device copies the image back to the host memory. (4) The GPU uploads the image to the GPU memory. However, with the new architecture, only two memory copies are needed: (1) The image is copied from the camera hardware to the host memory. (2) The SVM framework copies the image from the host memory to GPU memory.

Further, the SVM framework also opens up opportunities for special-case optimizations. For instance, when two virtual devices (*e.g.,* codec and GPU) share the same physical device (*e.g.,* GPU), coherence maintenance can be done in-GPU, without having to copy to and from host memory at all. Essentially, the unified architecture allows the virtual devices to find the shortest path for coherence maintenances and saves both time and bandwidth.

Meanwhile, through interacting with the virtual devices, SVM Manager collects SVM-related statistics across the system stack (from apps to hardware). As will be detailed in the next subsection (§3.3), 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 7, 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

[3]The image cannot be directly copied from the camera hardware to the guest memory, because camera APIs in OSes like Windows and macOS do not accept non-contiguous virtual memory (*e.g.,* guest memory) as destinations.

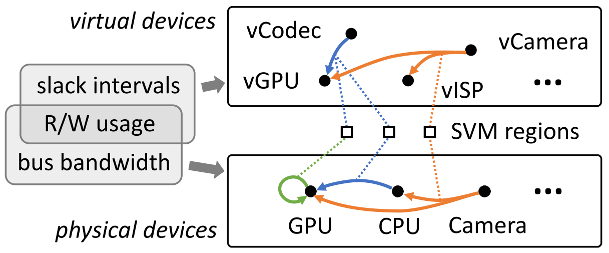


Figure 7. The structure of the twin hypergraphs. Each hyperedge characterizes a data flow. The dotted lines linking the two hypergraphs represent hashtable entries. The shaded areas to the left exemplify the data recorded in each hypergraph.

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 corresponding 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.

3.3 The 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 §2.4: it overlaps coherence maintenance with the slack intervals to save time. 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, which are built on the assumption that data sharing among devices is efficient on a mobile SoC. More concretely, high SVM access latency causes apps to miss the current frame deadline and wait for the next (>16 ms of waiting), and in turn causes system services to further miss the deadline by hundreds of milliseconds and lead to visible frame drops (see §5.4 for an example).

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 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 8 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. Suppose that the prefetch will take 10 *ms* to complete, and there will be an 8 *ms*

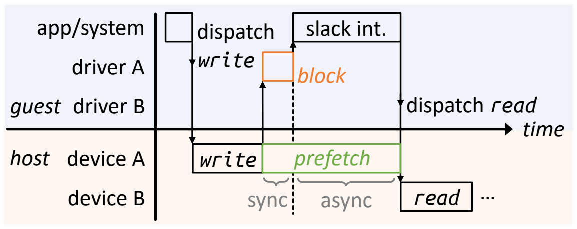


Figure 8. Timeline of the robust prefetch protocol.

slack interval between the last write and the next read. To fully utilize the slack intervals, prefetch begins immediately after device A finishes writing. Then, to avoid SVM access latency which may hurt app performance, 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, and may lead to performance loss.

To realize the robust prefetch protocol, some of the statistics (*e.g.,* the 10 *ms* and 8 *ms* in the above example) 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 (see §2.3). 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. For instance, while video apps may use different numbers of buffers for different videos, we do not observe any app that changes its buffering/VSync strategy during the playback of one video, *i.e.,* the slack intervals should be stable.

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 [19], one of the most widely used forecasting techniques due to its simplicity, robustness, and accuracy [80]. 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.

Finally, in corner cases when prediction failures happen three consecutive times, or the available bandwidth corresponding to the operation falls below 50% of the maximum observed bandwidth, we temporarily suspend prefetch to avoid bandwidth waste.

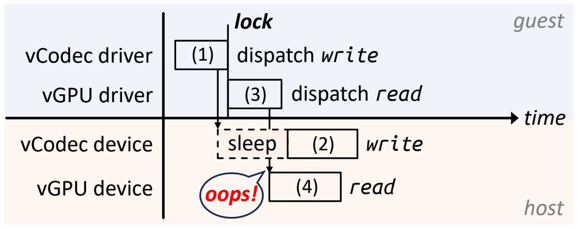
3.4 Virtual Command Fence

Similar to many mobile emulators [18, 28], the guest drivers and the host devices of vSoC are designed to work asynchronously most of the time: the guest driver dispatches control commands to the host virtual device, which has its own threads and does the actual data processing. Each device has one or more command queues, so that the guest can send asynchronous commands in batch to reduce transport overhead across the virtualization boundary. While the threading paradigm works fine with independent virtual devices, an access ordering problem arises when multiple virtual devices operate on shared resources such as shared memory.

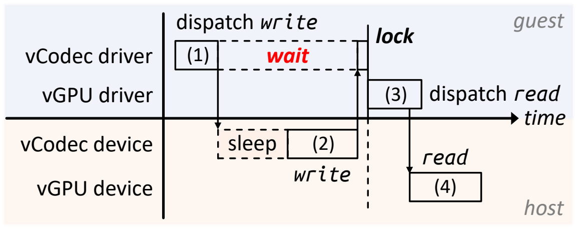
Take the common case of video rendering as an example. The ideal execution order is as follows: (1) The codec driver dispatches a `write` command to the virtual codec device to write a decoded video frame to an SVM region; (2) The virtual codec device writes to the SVM; (3) The guest OS forwards the 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 9a), since virtual devices are independently scheduled from guest drivers.

To solve the problem, a common approach [18, 28] is to let the guest driver perform shared resource operations *atomically*. As shown in Figure 9b, 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 introduces extra VM-Exits from interrupts.

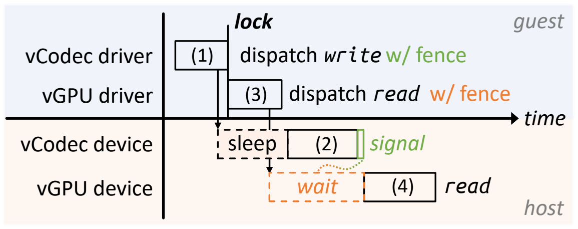
In essence, the key challenge to efficient access ordering is that command orders are only known to guest drivers, but need to be enforced in the host. Existing paradigms all rely on guest drivers to guarantee access ordering, leading to frequent guest-host control flow synchronizations that bring



(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 9. An example of access ordering between virtual devices.

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. To begin with, 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 [60], though multiple waits on a "signal" fence are also allowed. Figure 9c illustrates the usage of the fences: the codec driver inserts a "signal" fence into the command queue of the codec virtual device after Step 2, and the GPU driver inserts a "wait" fence before Step 4. 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 MMIO, while the physical fence tables are stored in the host.

Moreover, the virtual command fence mechanism is generic and can be used in other weak-state synchronization cases in which states in the host and guest should be eventually (but not strongly) consistent. For instance, the mechanism is also applied in GPU context switches to avoid GPU driver stalls which reduce throughput. The rationale is that GPUs are designed to be asynchronous from CPU: context switches in the GPU virtual device can be deferred so long as the order of switches is correct. Meanwhile, since the mechanism increases the asynchronism of execution between the guest and the host, and can cause guest drivers to send commands too quickly (since they do not have to wait for host execution anymore), we adopt the MIMD flow control algorithm of Trinity [59] to pace the execution of the two sides and avoid commands from piling up in host command queues.

4 System Implementation

vSoC is based upon QEMU 7.1 [72], and hosts Android-x86 9.0 [13], as well as OpenHarmony 4.0 [68]. 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 [34].

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` [40] protocol, a de-facto standard used both in macrokernels like Linux and microkernels like seL4 [43].

Regarding the internals of the SVM framework, the prefetch engine uses the DMA capabilities of supported devices (*e.g.*, GPUs) to help reduce CPU load. Additionally, since the virtual fence table used in §3.4 needs to be shared between the guest and the host, we limit its size to one memory page (usually 4 KiB) to avoid the overhead of accessing non-contiguous guest pages in the host, and we recycle signaled fence indices when the supply of unused indices is low.

Besides an efficient SVM framework, a solid implementation of the virtual devices is also necessary for the efficiency of vSoC. To leverage the high-performance virtual GPU of the Trinity emulator [18], 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` [20], a cross-platform library for window management. The virtual ISP (Image Signal Processor) utilizes Google Android Emulator's `YUVConverter` module [5] for in-GPU colorspace conversion of certain image formats, and `libswscale` [54], a software colorspace conversion library for other formats.

The virtual codec is implemented with `libavcodec` [53], a popular library that supports software and hardware decoding/encoding. We further use the OpenGL interop extensions [66] to achieve in-GPU video rendering with popular formats. The guest codec driver is implemented against the OpenMAX IL specification [42] as required by Android and OpenHarmony. The virtual camera is implemented with `libavdevice`, a cross-platform library for manipulating peripheral devices. The virtual cellular modem is implemented with reference to Google Android Emulator's `telephony` module [26] and supports the Radio Interface Layer (RIL) [30] of Android and the RIL adapter [69] of OpenHarmony.

5 Evaluation

In this section, we answer the following questions about the performance of vSoC: (1) how does the SVM framework perform in practice (§5.2), (2) how does vSoC perform with five types of emerging mobile apps (§5.3), (3) what is the contribution of individual designs (§5.4), and (4) how do top popular mobile apps receive the performance benefits (§5.5).

5.1 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 (§2.3), 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, 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.

Table 2. SVM performance on the two PCs (high-end desktop vs. middle-end laptop).

Metric	vSoC	GAE	QEMU-KVM
Access Latency	0.34 / 0.38 *ms*	0.76 / 1.16 *ms*	0.22 / 0.25 *ms*
Coherence Cost	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

Target Emulators. Apart from vSoC, five mainstream mobile emulators are involved in the evaluation, including Google Android Emulator (GAE) [28], QEMU-KVM [73], LDPlayer [48], Bluestacks [65], and Trinity [18]. The latest publicly-available versions of the emulators as of Mar. 2024 are used. 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 emerging mobile platforms (*e.g.,* TVs and AR/VR headsets), which are usually equipped with high-resolution displays [64] including QHD, UHD, and even 8K.

Workloads. The apps involved are the 50 emerging apps listed in Table 1, and the workloads are the same as those in §2.3. Each app is tested for 5 minutes on each emulator.

5.2 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 §2.3 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. Since source code instrumentation is needed, only GAE and QEMU-KVM are involved for comparison.

Results. Table 2 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 §2.3), the average SVM throughput of vSoC is much higher than those of both GAE and QEMU-KVM (163% and 264% higher, respectively).

The predictions involved in the prefetch coherence protocol turn out to be very accurate: we observe that its device prediction accuracy varies between 99% and 100% in the five categories. The predictions of slack intervals and prefetch time cost have low standard errors of 0.9 *ms* and 0.3 *ms* respectively. A manual inspection of the failures reveals that the prediction errors mostly occur during emulator startup, when no historical data are available in the twin hypergraphs. Benefiting from the robust prefetch protocol, coherence maintenance seldom leads to SVM access latency, and the average access latency of vSoC is a negligible 0.3 *ms*, only slightly higher than the access latency of QEMU-KVM due to the overhead of misprediction. The access latency of QEMU-KVM is the lowest since its SVM is based on guest memory and only involves page mapping costs, but that is at the expense of both coherence time and throughput.

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.

5.3 Application Benchmarks

Methodology. The FPS and motion-to-photon latency of the emerging apps are measured. The two metrics are key indicators of the smoothness and responsiveness of mobile apps [35]. 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 [23]. 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. We position the camera at the side of the computer screen, so that both user actions and the screen can be clearly observed. To ease recognition of user actions, for camera and AR apps, we use flashlights to produce sudden luminance changes on camera streams. For livestream apps, we flash the screen contents of the emulators using the built-in developer tools of Android [25]. 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 emerging 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) [21] 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 two PCs. On the high-end machine, vSoC can achieve nearly full (57) FPS on all five types of emerging 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 emerging 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 [18], it performs badly when running emerging 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 emerging apps in the UHD/360 Video category, because Trinity only has a software virtual codec device inherited from Android-x86 [13]. Since Trinity's focus is on GPUs, it does not implement hardware 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 within 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). The results further demonstrate the benefits of a fully paravirtualized architecture as opposed to hybrid architectures: while common wisdom suggests that PCIe pass-through is efficient and low-overhead [71], pass-through devices introduce VM transport overhead. When sharing data with paravirtualized devices, making it inferior to a fully paravirtualized solution like vSoC. On the middle-end laptop, the end-to-end latency

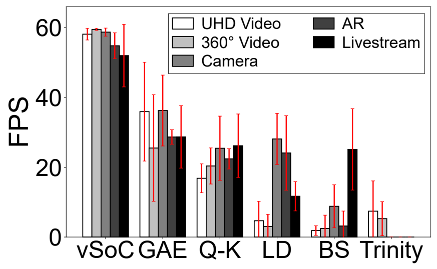


Figure 10. FPS on high-end PC.

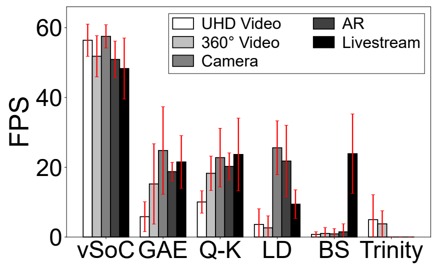


Figure 11. FPS on middle-end PC.

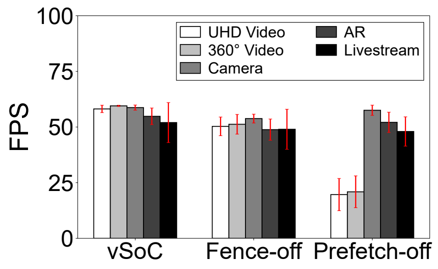


Figure 12. FPS breakdown on high-end PC.

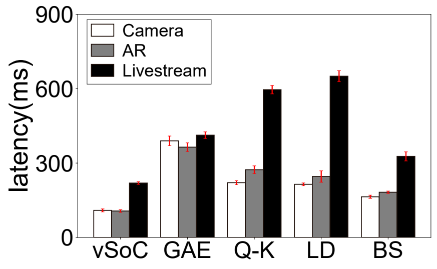


Figure 13. Latency on high-end PC.

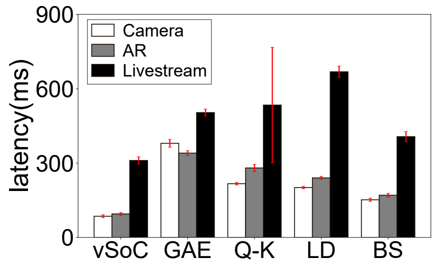


Figure 14. Latency on middle-end PC.

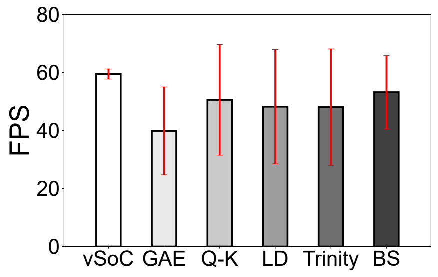


Figure 15. FPS of other apps on high-end PC.

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 [76] of Windows, and find out that the end-to-end latency of the integrated camera is indeed 10 *ms* lower than the USB camera of the high-end desktop.

To prevent the bias introduced by apps that cannot be run on some emulators, we further make a pairwise comparison between vSoC and each of the mainstream emulators in terms of the FPS and latency of the apps (that vSoC and the compared emulator can both successfully run). The results are very close to the performance indicated by the bar plots: compared to the other emulators, vSoC can achieve 84%-828% better FPS and 36%-65% less latency on average on the high-end machine, and 192%-1134% better FPS and 34%-64% less latency on the middle-end machine.

Qualitative Study. We conducted a user study in Huawei to understand the user-perceivable differences on the performance of the emerging apps in Apr. 2024. 90 mobile developers from Huawei (aged 20-60, average 31) participated in the study. We set up each emulator in a dedicated middle-end laptop (with the same configuration as §5.1), and allow the user to freely use the emerging apps and switch emulators for a ten-minute session. The emulators' identities are hidden from the users. In the meantime, they are asked to fill in a questionnaire that ranks and gives ratings to the emulators on video smoothness perceived by eyes, motion-to-photon latency on camera/AR streams, and their overall user experience. The study is conducted with informed consent in accordance with the ethical guidelines of the IRB.

In summary, the study reveals that most users do perceive noticeable performance improvements in vSoC, especially for experienced users of emerging 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 emerging apps like AR/VR require 60-180 FPS and sub-100ms motion-to-photon latency to achieve a satisfying user experience and avoid motion sickness [47, 86].

5.4 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 [36] for coherence management instead. In the write-invalidate protocol, memory is lazily updated at the beginning of each SVM access (in the

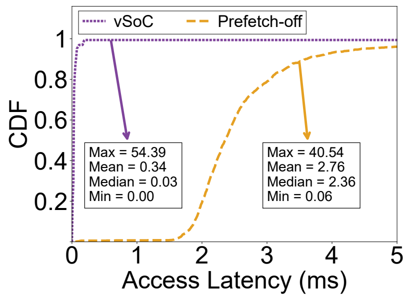


Figure 16. Access latency on high-end PC when the prefetch engine is turned off.

`begin_access` API of §2.1) to reduce memory bandwidth consumption. Since data will be accessed by the new device after the API returns, coherence maintenance needs synchronous guest-host execution, and thus virtual command fences cannot be used (other usages of the fences are not touched). Second, when the virtual fence mechanism is disabled, we fall back to commonly-adopted atomic operations (see §3.4) for access ordering among host threads.

Results. As shown in Figure 12, after the prefetch engine is turned off, the average performance of the emerging 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, as shown in Figure 16. To achieve real-time playback and avoid stale frames, the video renderer (`MediaCodec` [24]) 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 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 §3.4).

5.5 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 emerging ones, we conduct regression testing on the top-25 popular apps in Google Play (as of Apr. 2024), and record the FPS of the apps. The measurement methodology is the same as the application benchmarks in §5.3. Moreover, we also carry out the performance breakdown experiment in §5.4 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, so we further conduct a pairwise FPS comparison between vSoC and each of the mainstream emulators, using data from the apps that vSoC and the compared emulator can both 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 [31]) 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.

6 Porting to vSoC

vSoC already provides paravirtualized implementations of a common set of SoC devices that satisfy the needs of most mobile apps on the market. For those who want to add new virtual devices to vSoC, they can choose to either port the virtual devices into the SVM framework or leave them as-is. vSoC is compatible with existing I/O virtualization approaches like PCIe pass-through (vSoC shares memory with the new virtual device via the traditional method in §2.2), so the new virtual device will work without modification and without performance penalties. But to enjoy the performance benefits of vSoC, one has to port the virtual device into the SVM framework. Fortunately, porting virtual devices to vSoC is usually a tractable process.

For common devices like GPU and camera (which already have paravirtualized drivers), the porting process is quite manageable. Usually, only memory-related code needs to be changed, and memory-management functions usually account for less than 20% of all the driver functions [37]. Specifically, the virtual device (along with its driver) needs to provide a handle representation of its local memory, feed information of its SVM usage into the hypergraphs, add prefetch and fence commands after SVM accesses, and finally, provide means of copying data from and to other physical devices. As a reference, GAE's image processor driver and Trinity's GPU driver can be minimally ported to vSoC respectively with ~150 and 4K lines of code diffs.

If the virtual device does not have a paravirtualized driver, porting to vSoC will require converting to a paravirtualized driver, since the SVM framework is paravirtualized. Certainly, the engineering effort is not negligible. Thus, it is recommended to prioritize porting drivers of high-throughput devices. For devices that are not performance-critical or data-intensive (*e.g.*, keyboards or touchscreens), conventional I/O virtualization like PCIe pass-through can be used instead.

7 Related Work

Efficient Mobile Emulation. Mobile emulation has become a keystone of the mobile ecosystem and spans many areas of interest including mobile app development [14] and testing [55], malware detection [4], personal gaming [84], and cloud gaming [52, 79]. Therefore, much effort from both academia and industry has been put in to enable efficient emulation of mobile systems.

An important line of work seeks to improve the I/O virtualization efficiency of individual mobile hardware, including CPU [15, 73], GPU [16, 18, 84, 85], NIC [46, 74], and display [7, 12, 58]. In contrast, our work does not intend to further improve the efficiency of individual devices, but stands from the perspective of a *system* of virtual devices and improves the efficiency of their interactions. That is particularly important for emerging mobile apps designed to fully tap the performance potential of mobile SoCs.

Besides, there has been recent work running mobile systems in containers [11, 12, 52] or by pruning or multiplexing mobile system services [79] to reduce the virtualization overhead. Since they only enable the sharing of mobile system components or the Linux kernel but not heterogeneous PC/server hardware, our work is largely orthogonal to them.

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 [3, 8, 9, 17, 39, 50, 62] that designs robust memory management strategies for heterogeneous memory systems, compiler enhancements [75, 77] that provide transparent SVM implementations for programs, and hardware integration [45, 63, 82, 87] 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 where the architectural gap between mobile devices and PC/server systems poses unique challenges (see §2.2).

More detailedly, regarding the choice of coherence protocols, many SVM implementations adopt classical write-invalidate [38] or broadcast [44, 78] protocols, which are unsuitable in mobile emulation because of high access latency or bandwidth overhead. Some works explore software [10] and hardware prefetching [3, 17] for coherence management, but the proposed prefetch protocols typically face a tradeoff of prefetch aggressiveness. Our work instead circumvents the tradeoff by exploiting the usage characteristics of shared memory in mobile systems.

Meanwhile, regarding the granularity of SVM memory units, most SVM related work [3, 17, 51, 87] adopts a fixed page-level unit size, whereas mainstream mobile emulators often segment SVM into smaller memory units according to the size of the dirty region provided by the shared memory API (see Figure 3), because processing large numbers of page faults across the virtualization boundary can bring significant VM-Exit [18, 81] overhead that stalls the whole system. vSoC also adopts the common practice of mainstream mobile emulators, though our core mechanisms can apply to other SVM unit granularities as well.

8 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 emerging mobile 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.

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

We are very grateful to our shepherd, Jason Nieh, for his invaluable feedback. We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703, the National Natural Science Foundation of China under grants 62332012 and 62472245, and the Ant Group. Tianyin Xu is supported by NSF CNS-1956007.

References

[1] Adobe Systems. Adobe RTMP Specification, 2019. https://rtmp.veriskope.com/docs/spec/.

[2] Jasmin Ajanovic. PCI Express 3.0 Overview. In *Proc. of IEEE HCS*, pages 1–61, 2009.

[3] Tyler Allen and Rong Ge. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *Proc. of ACM/IEEE SC*, pages 1–15, 2021.

[4] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning. In *Proc. of ACM IWSPA*, pages 65–72, 2017.

[5] 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.

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

[7] Ricardo A. Baratto, Shaya Potter, Gong Su, and Jason Nieh. MobiDesk: Mobile Virtual Desktop Computing. In *Proc. of ACM MobiCom*, pages 1–15, 2004.

[8] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of ACM PPoPP*, pages 168–176, 1990.

[9] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proc. of ACM ASPLOS*, pages 79–92, 2021.

[10] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. *ACM SIGARCH Computer Architecture News*, 19:40–52, 1991.

[11] Canonical. Anbox Cloud Official Website, 2024. https://anbox-cloud.io.

[12] Wenzhi Chen, Lei Xu, Guoxi Li, and Yang Xiang. A Lightweight Virtualization Solution for Android Devices. *IEEE Transactions on Computers*, 64:2741–2751, 2015.

[13] Chi-Wei Huang. Android-X86 Release 9.0-R2, 2024. https://www.android-x86.org/releases/releasenote-9-0-r2.html.

[14] Jaewon Choi, Seungchan Jeong, and JeongGil Ko. Emulating Your eXtended World: An Emulation Environment for XR App Development. In *Proc. of IEEE MASS*, pages 131–139, 2022.

[15] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low Overhead Dynamic Binary Translation on ARM. In *Proc. of ACM PLDI*, pages 333–346, 2017.

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

[17] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proc. of ACM/IEEE ISCA*, pages 224–235, 2019.

[18] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity: High-Performance Mobile Emulation through Graphics Projection. In *Proc. of USENIX OSDI*, pages 285–301, 2022.

[19] Everette S. Gardner Jr. Exponential Smoothing: The State of the Art. *Journal of Forecasting*, 4:1–28, 1985.

[20] GLFW Developers. GLFW: An OpenGL Library, 2024. https://www.glfw.org/.

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

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

[23] Google. Android Debug Bridge, 2024. https://developer.android.com/tools/adb.

[24] Google. Android MediaCodec, 2024. https://developer.android.com/reference/android/media/MediaCodec.

[25] Google. Android On-Device Developer Options, 2024. https://developer.android.com/studio/debug/dev-options.

[26] Google. Android Telephony Android Developers Documents, 2024. https://developer.android.com/reference/android/telephony/package-summary.

[27] Google. ARCore Official Website, 2024. https://developers.google.com/ar.

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

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

[30] Google. RIL Refactoring, 2024. https://source.android.com/docs/core/connect/ril.

[31] Google. Skia Official Website, 2024. https://skia.org/.

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

[33] HUAWEI. HUAWEI DevEco Studio, 2024. https://devecostudio.huawei.com/en/.

[34] 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.

[35] Michael Jarschel, Daniel Schlosser, Sven Scheuring, and Tobias Hoßfeld. An Evaluation of QoE in Cloud Gaming Based on Subjective Tests. In *Proc. of Springer IMIS*, pages 330–335, 2011.

[36] Norman P. Jouppi. Cache Write Policies and Performance. *ACM SIGARCH Computer Architecture News*, 21:191–201, 1993.

[37] Asim Kadav and Michael M. Swift. Understanding Modern Device Drivers. *ACM SIGPLAN Notices*, 47:87–98, 2012.

[38] Abdullah Kayi, Olivier Serres, and Tarek El-Ghazawi. Adaptive Cache Coherence Mechanisms with Producer–Consumer Sharing Optimization for Chip Multiprocessors. *IEEE Transactions on Computers*, 64:316–328, 2015.

[39] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy 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*, page 10, 1994.

[40] Kernel Developers. Virtio on Linux — The Linux Kernel Documentation, 2024. https://docs.kernel.org/driver-api/virtio/virtio.html.

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

[42] Khronos. OpenMAX Official Website, 2011. https://www.khronos.org/openmax/.

[43] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of ACM SOSP*, pages 207–220, 2009.

[44] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *Proc. of ACM/IEEE ISCA*, pages 996–1009, 2020.

[45] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michał Cierniak, Srinivasan Parthasarathy, Wagner Meira, Sandhya Dwarkadas, and Michael Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. *ACM SIGARCH Computer Architecture News*, 25:157–169, 1997.

[46] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *Proc. of ACM SIGCOMM*, pages 351–366, 2019.

[47] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *Proc. of ACM MobiCom*, pages 409–421, 2017.

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

[49] 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.

[50] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of ICPP*, 1988.

[51] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7:321–359, 1989.

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

[53] Libavcodec Developers. Libavcodec Documentation, 2024. https://www.ffmpeg.org/libavcodec.html.

[54] Libswscale Developers. Libswscale Documentation, 2024. https://ffmpeg.org/libswscale.html.

[55] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility. In *Proc. of ACM MobiCom*, pages 1–17, 2023.

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

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

[58] Yan Lu, Shipeng Li, and Huifeng Shen. Virtualized Screen: A Third Element for Cloud–Mobile Convergence. *IEEE MultiMedia*, 18:4–11, 2011.

[59] Stephen F. Lundstrom. Controllable Multiple-Instruction, Multiple-Data Stream (MIMD) Architecture. In *Real-Time Signal Processing V*, pages 333–338, 1982.

[60] Pallavi Maiya and Aditya Kanade. Efficient Computation of Happens-before Relation for Event-Driven Programs. In *Proc. of ACM ISSTA*, pages 102–112, 2017.

[61] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race Detection for Android Applications. *ACM SIGPLAN Notices*, 49:316–325, 2014.

[62] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs. *Proceedings of the VLDB Endowment*, 14:114–127, 2020.

[63] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8:12–20, 2000.

[64] Michael Nebeling, Shwetha Rajaram, Liwei Wu, Yifei Cheng, and Jaylin Herskovitz. XRStudio: A Virtual Production and Live Streaming System for Immersive Instructional Experiences. In *Proc. of ACM CHI*, pages 1–12, 2021.

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

[66] NVIDIA. CUDA OpenGL Interoperability, 2024. https://docs.nvidia.com/cuda/cuda-driver-api/index.html.

[67] OpenHarmony. OpenHarmony Official Website, 2024. https://gitee.com/openharmony.

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

[69] OpenHarmony. OpenHarmony telephony_ril_adapter Repository, 2024. https://gitee.com/openharmony/telephony_ril_adapter?_from=gitee_search.

[70] Mahendra PratapSingh and Manoj Kumar Jain. Evolution of Processor Architecture in Mobile Phones. *International Journal of Computer Applications*, 90:34–39, 2014.

[71] Project ACRN™ documentation. Device Passthrough, 2024. https://projectacrn.github.io/latest/developer-guides/hld/hv-dev-passthrough.html.

[72] QEMU Developers. QEMU Version 7.1.0, 2022. https://www.qemu.org/2022/08/30/qemu-7-1-0/.

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

[74] Himanshu Raj and Karsten Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proc. of ACM HPDC*, pages 179–188, 2007.

[75] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of ACM ASPLOS*, pages 174–185, 1996.

[76] stevewhims. DirectShow - Win32 Apps, 2023. https://learn.microsoft.com/en-us/windows/win32/directshow/directshow.

[77] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation. In *Proc. of ACM PLDI*, pages 329–345, 2020.

[78] Weiyi Sun, Zhaoshi Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-Based Near-Memory Processing with Inter-DIMM Broadcast. In *Proc. of ACM/IEEE ISCA*, pages 237–250, 2021.

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

[80] Sean J. Taylor and Benjamin Letham. Forecasting at Scale. *The American Statistician*, 72:37–45, 2018.

[81] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38:48–56, 2005.

[82] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *Proc. of USENIX FAST*, pages 277–292, 2021.

[83] Will Reese. Nginx: The High-Performance Web Server and Reverse Proxy, 2008. https://dl.acm.org/doi/fullHtml/10.5555/1412202.1412204.

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

[85] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated Virtualization of Accelerators. In *Proc. of ACM ASPLOS*, pages 807–825, 2020.

[86] Wenxiao Zhang, Bo Han, and Pan Hui. SEAR: Scaling Experiences in Multi-User Augmented Reality. *IEEE Transactions on Visualization and Computer Graphics*, 28:1982–1992, 2022.

[87] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L. Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs. *ACM Transactions on Architecture and Code Optimization*, 13:35:1–35:25, 2016.