Trinity: High-Performance and Reliable Mobile Emulation through Graphics Projection

HAO LIN, ZHENHUA LI, DI GAO, and YUNHAO LIU, Tsinghua University, Beijing, China
FENG QIAN, University of Minnesota, Minnesota, USA
TIANYIN XU, University of Illinois Urbana-Champaign, Illinois, USA
BO XIAO and XIAOKANG QIN, Ant Group, Hangzhou, China

Mobile emulation, which creates full-fledged software mobile devices on a physical PC/server, is pivotal to the mobile ecosystem. Unfortunately, existing mobile emulators perform poorly on graphics-intensive apps in terms of efficiency and compatibility. To address this, we introduce *graphics projection*, a novel graphics virtualization mechanism that adds a small-size *projection space* inside the guest memory, which processes graphics operations involving control contexts and resource handles without host interactions. While enhancing performance, the decoupled and asynchronous guest/host control flows introduced by graphics projection can significantly complicate emulators' reliability issue diagnosis when faced with a variety of uncommon or non-standard app behaviors in the wild, hindering practical deployment in production. To overcome this drawback, we develop an automatic reliability issue analysis pipeline that distills the critical code paths across the guest and host control flows by runtime quarantine and state introspection. The resulting new Android emulator, dubbed Trinity, exhibits an average of 97% native hardware performance and 99.3% reliable app support, in some cases outperforming other emulators by more than an order of magnitude.

CCS Concepts: • **Software and its engineering** → **Virtual machines**; *Software testing and debugging*; • **Computing methodologies** → **Graphics systems and interfaces**; • **Human-centered computing** → **Mobile computing**; • **Computer systems organization** → *Reliability;*

Additional Key Words and Phrases: mobile emulation, virtualization, graphics interfaces

ACM Reference Format:
Hao Lin, Zhenhua Li, Di Gao, Yunhao Liu, Feng Qian, Tianyin Xu, Bo Xiao, and Xiaokang Qin. 2024. Trinity: High-Performance and Reliable Mobile Emulation through Graphics Projection. *ACM Trans. Comput. Syst.* 42, 3–4, Article 6 (September 2024), 33 pages. https://doi.org/10.1145/3643029

This article is an extension of its prior conference version [27] appearing in the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22). The novel content lies mainly in our observation, diagnosis, and resolution on the reliability problems of the high-performance Trinity emulator. In particular, we devise a root cause analysis method for debugging cross-layer reliability issues, as well as address the issues via manifold extensions and enhancements of Android's standard OpenGL ES graphics framework. In addition, owing to better workflow synchronization and resource saving brought by the enhancements, Trinity even gains an average of 26% improvement in efficiency.

This work is supported in part by National Key R&D Program of China under grant 2022YFB4500703, National Natural Science Foundation of China under grant 62332012, and the Ant Group.

Authors' addresses: H. Lin, Z. Li (Corresponding author), D. Gao, and Y. Liu, Tsinghua University, Beijing 100084, China; e-mails: linhaomails@gmail.com, lizhenhua1983@gmail.com, gaodi.sec@gmail.com, yunhaoliu@gmail.com; F. Qian, University of Minnesota, Minnesota 55455, USA; e-mail: fengqian@umn.edu; T. Xu, University of Illinois Urbana-Champaign, Illinois 61825, USA; e-mail: tyxu@illinois.edu; B. Xiao and X. Qin, Ant Group, Hangzhou 310012, China; e-mails: xiaobo.xiao@antgroup.com, xiaokang.qxk@antgroup.com.

ACM 0734-2071/2024/09-ART6
https://doi.org/10.1145/3643029

1 INTRODUCTION

Mobile emulation has been a keystone of the mobile ecosystem. Developers today typically debug their apps on generic mobile emulators (e.g., Google's Android Emulator, or GAE for short) rather than on heterogeneous real devices. Also, various dedicated mobile emulators (e.g., Bluestacks [15] and DAOW [80]) are used to detect malware in app markets [29, 63, 79], to enable mobile gaming on PCs [15, 80], and to empower the emerging notion of cloud gaming [49].

Motivation. To create software mobile devices on a physical PC/server, mobile emulators usually adopt the classic virtualization framework [44, 57, 64, 66] where a mobile OS runs in a **virtual machine (VM)** (i.e., the guest), hosted on a PC/server (i.e., the host). However, traditional virtualization techniques are initially designed to work on headless servers or common PCs without requiring strong UI interactions within the VM, while real-world mobile apps are highly interactive [50] and thus expecting mobile emulators to have powerful graphics processing capabilities (as provided by real mobile phones) [80]. This *capability gap* is further aggravated by the substantial architectural differences between the graphics stacks of desktop and mobile OSes [16].

Over the years, several approaches have been proposed to fill the gap. Perhaps the most intuitive is solely relying on a CPU to carry out a GPU's functions. For example, as a user-space library residing in mobile OSes (e.g., Android), SwiftShader [35] helps a CPU mimic the processing routines of a GPU. This achieves the best compatibility since any mobile app can thus seamlessly run under a wide variety of environments even without actual graphics hardware, but at the cost of poor efficiency since a CPU is never suited to handling the highly parallel (graphics) rendering tasks.

To improve the emulation efficiency, a natural approach is multiplexing the host GPU within a PC/server through API remoting [22, 72], which intercepts high-level graphics API calls at the guest and then executes them on the host GPU with dedicated RPC protocols and guest-host I/O pipes. Unfortunately, the resulting products (e.g., GAE) cannot smoothly run many common apps, let alone "heavy" (i.e., graphics-intensive) apps for AR/VR viewing and 3D gaming. This shortcoming stems from frequent VM Exits to the host to execute API calls, introducing a considerable "tromboning" effect [25] on the control and data flows. This results in additional idle waiting at the guest, as it must wait not only for the API call to complete, but also for the added process of exiting to the host and returning back to the guest.

To mitigate the issue, *device emulation* [20] moves the virtualization boundary from the API level to the driver level. It forwards guest-side graphics driver commands to the host with a shared memory region inside the guest kernel to realize their effects with the host GPU. Compared to high-level APIs, driver commands are much fewer, more capable, and mostly asynchronous [20], so device emulation effectively reduces guest-host control/data exchanges and idle waiting. However, the translation from API calls to driver commands degrades critical high-level abstractions such as windows and threads to low-level memory addresses and register values. Due to the loss of high-level information, driver commands must be sequentially executed at the host, degrading guest-side multi-threaded rendering to host-side single-threaded rendering. Hence, the resulting emulators (e.g., QEMU-KVM) can smoothly run regular apps but not heavy ones.

Another approach is to break guest-host isolation by removing the virtualization layer so apps can directly use the GPU, as embodied in DAOW [80]. This requires manually translating Linux system calls used by Android to Windows ones. Unfortunately, many apps cannot run on DAOW because many (~46%) system calls are not translated due to the huge engineering efforts required for full system calls' translation. Also, the supported apps must run under the protection of additional sophisticated security defenses to compensate for the lack of guest-host isolation.

We present Trinity, a novel mobile emulator that simultaneously achieves high efficiency, compatibility, and reliability. Our guiding principle is to decouple the guest-host control and data

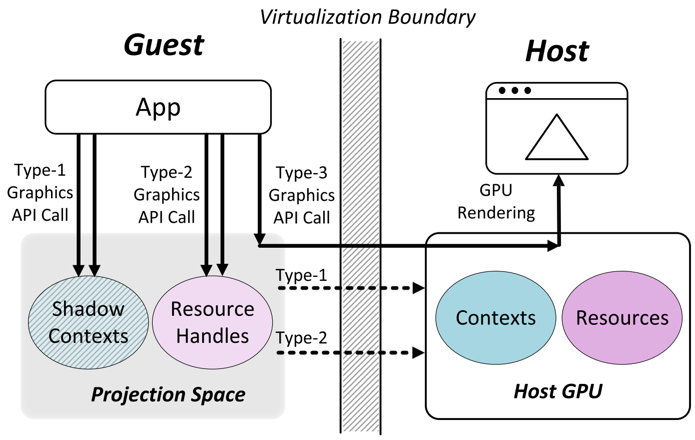


Fig. 1. Basic workflow of Trinity.

exchanges and make them as asynchronous as possible when multiplexing the host GPU under the virtualization framework, so that frequent VM Exits for synchronous host-side execution of API calls can be largely reduced. For this purpose, we propose to add a *projection space* inside the guest memory, where we selectively maintain a "projected" subset of control contexts (termed *shadow contexts*) and resource handles. Such contexts and handles are derived but different from the real ones required by a physical GPU to perform rendering, so as to reflect and reproduce the effects of guest-side graphics operations (i.e., API calls). Thus, the vast majority (99.93%) of API calls do not need synchronous execution at the host, while consuming less than 1 MB memory for even a heavy 3D app.

Concretely, when an Android app wants to draw a triangle on a physical phone, it sequentially issues three types of graphics API calls: context setting (Type-1), resource management (Type-2), and drawing (Type-3). Type-1 prepare the canvas and bind resource handles; Type-2 populate the handles' underlying resources with the triangle's vertex coordinates, filling colors/patterns, *and so on*; Type-3 instruct the GPU to render and display the triangle. In contrast, as shown in Figure 1, when the app runs in Trinity, Type-1 and Type-2 calls are first executed only in the projection space, that is, their effects are temporarily reflected on the shadow contexts and resource handles. Later upon drawing calls (Type-3), their effects are delivered to the host to realize actual rendering.

Combined with graphics projection, an elastic flow control algorithm is devised to orchestrate the control flows at both the guest and host sides. Regarding the guest-host data flows, we find that the major challenge of rapidly delivering them lies in the high dynamics of system status and data volume (e.g., bursty data flows common in graphics workloads). To this end, we find that the dynamic situations follow only a few patterns, each of which requires specific data aggregation, persistence, and arrival notification strategies. Therefore, we implement all the required strategies, and utilize *static timing analysis* [13] to estimate which strategy is best suited to a data flow.

While graphics projection is able to decouple the guest and host control flows to enhance performance, it also significantly increases the complexity of emulators' bug analysis, especially when facing a variety of reliability issues (e.g., functional failures, app crashes, and system hangs) incurred by uncommon, non-standard, or even erroneous behaviors of mobile apps in the wild. Under the graphics projection framework, reliability issues usually manifest in a cross-layer manner, that is, the symptoms and code paths are often distinct and hard to correlate at the guest and host sides, making the debugging space prohibitively large. Without an effective method to diagnose and debug, the deployment of graphics projection in production is extremely difficult.

To tackle the abovementioned difficulty and benefit other modern virtualization systems which widely adopt cross-layer design (e.g., paravirtualization-based device emulation) for optimizing the virtualization efficiency, we develop an automatic analysis pipeline to effectively diagnose cross-layer reliability issues. Our key idea is to capture host-side and guest-side rendering code flows responsible for a reliability issue at run time, and then "quarantine" them as standalone programs that can help us effectively reproduce the issue. Concretely, we compare the host-side runtime states (i.e., control contexts and graphics resources) with those of the guest app; the uncovered state differences then act as key indicators of Trinity's internal bugs. This is enabled by Trinity's role of graphics API provider and the projection space that manages guest-side runtime states, which make guest apps' API calls and relevant states highly introspectable, even without the apps' source code. Thereby, we can distill the critical code paths (that essentially lead to reliability issues) by strategically altering the parameters of the API calls related to the states and checking whether the failure scene is affected in the meantime. With this root cause analysis method, we are able to efficiently pinpoint the root causes of all the reliability issues of Trinity reported during its beta testing before production deployment, and effectively resolve all of them.

Evaluation. Similar to GAE, Trinity is also implemented atop QEMU and hosts the Android OS, with 120K lines of C/C++ code. We evaluate its efficiency, compatibility and reliability using standard graphics benchmarks, the top-100 3D apps from Google Play, and 10K apps randomly selected from Google Play. We also compare the results with six mainstream emulators: GAE, QEMU-KVM, **Windows Subsystem for Android** (**WSA**), VMware Workstation, Bluestacks, and DAOW. The evaluation shows that Trinity can achieve 75%~111% (averaging at 97%) native hardware performance, outperforming the other emulators by 1.4× to 20×. For compatibility and reliability, Trinity can run the top-100 3D apps and 99.3% of the 10K randomly selected apps. To our knowledge, Trinity is the first and the only Android emulator that can smoothly run heavy 3D apps without losing compatibility.

Software/Code/Data Availability. The binary, code, and measurement data involved in this work are released at https://TrinityEmulator.github.io/.

2 UNDERSTANDING MOBILE GRAPHICS API

We first delve into the three types of APIs in OpenGL ES, the de facto graphics framework of Android (Section 2.1), and then measure real-world 3D apps to obtain an in-depth understanding of their graphics workloads (Section 2.2).

2.1 Background

Figure 2 shows a basic OpenGL ES program for drawing a triangle. The program first creates a graphics buffer in a GPU's graphics memory using a Type-2 API—`glGenBuffers`, then populates the buffer with the coordinate data of the triangle's vertices through a Type-1 API—`glBindBuffer` and a Type-2 API—`glMapBufferRange`, and finally instructs the GPU to draw the triangle using a Type-3 API—`glDrawArrays`.

Type-1: Context Setting. To manipulate or use the allocated graphics buffer, instead of passing the buffer's handle to every API call, the program first calls `glBindBuffer`, which binds the handle to a thread-local *context*, that is, the transparent, global state of the thread. Then, all the subsequent buffer-related API calls (e.g., the buffer population call `glBufferData` and the drawing call `glDrawArrays` that uses the buffer data to draw) will be directly applied to the bound buffer, without needing to specify the buffer handle in their call parameters.

```
float vertices[9] = {  0.0f,  0.5f,  0.0f, // First vertex
                     -0.5f, -0.5f,  0.0f, // Second vertex
                      0.5f, -0.5f,  0.0f  // Third vertex
};  // Triangle vertices' (x, y, z) coordinates

float *vtx_mapped_buf; // Address of the mapped buffer

void populate_buffer() {
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                                0, GL_DYNAMIC_DRAW);
    ...
    // Type-2: query the buffer size
    int buf_size;
    glGetBufferParameteriv(GL_ARRAY_BUFFER,
                GL_BUFFER_SIZE, &buf_size);
    // Type-2: map the buffer to main memory space
    vtx_mapped_buf = glMapBufferRange(GL_ARRAY_BUFFER,
                      0, buf_size, GL_MAP_WRITE_BIT);

    memcpy(vtx_mapped_buf, vertices, buf_size);
    // Type-2: unmap the buffer
    glUnmapBuffer(GL_ARRAY_BUFFER);
}
```

(a) Populate the graphics buffer by latent mapping.

```
uint vertex_buffer_handle; // Graphics buffer handle

void draw() {
    ...
    // Type-2: allocate a buffer and generate its handle
    glGenBuffers(1, &vertex_buffer_handle);
                  1. The buffer's handle is bound to the context
    // Type-1: bind the buffer to context
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_handle);

    populate_buffer();                            2. Subsequent
    ...                                          operations do not
    // Type-3: draw the triangle                 need to specify the
    glDrawArrays(GL_TRIANGLES, 0, 3);             handle again
    ...
}
```

(b) Draw the triangle.

Fig. 2. OpenGL ES code snippet for drawing a triangle.

The above process is called *context setting*, which configures critical information of the current thread's context. This programming paradigm avoids repeatedly transferring context information from the main memory to the GPU, particularly when the information is rarely modified. In general, the context information that requires setup includes the current *operation target*, *render configurations*, and *resource attributes*. The operation target identifies the object that subsequent API calls will affect, for example, in Figure 2 the buffer handle becomes the operation target of subsequent API calls after it is bound to the context. Render configurations define certain rendering behaviors, for example, whether to perform validation of pixel values after a frame is rendered. Resource attributes correspond to resources' internal information, for example, formats of images and data alignment specifications.

Type-2: Resource Management. Resources involved in graphics rendering include *graphics buffers* that store vertice and texture data (*"what to draw"*), *shader programs* that produce special graphics effects such as geometrical transformation (*"how to draw"*), and *sync objects* that set time-wise sync points (*"when to draw"*). Graphics buffers hold most of the graphics data and thus require careful management. To populate a buffer with graphics data, there are mainly two approaches—*immediate copy* and *latent mapping*.

With regard to immediate copy, data are passed into the `glBufferData` API's third call parameter and copied from the main memory to the bound graphics buffer, that is, the buffer pointed by `vertex_buffer_handle`. This approach is easy to implement but involves synchronous, time-consuming memory copies. In contrast, Figure 2(a) shows the latent mapping approach, where `glBufferData` is called but no data are passed to it; instead, `glMapBufferRange` maps the graphics buffer to a main memory address, that is, `vtx_mapped_buf`. The data can then be directly stored in the mapped main memory space, without needing to synchronously trigger memory-to-GPU copies. Finally, when the data are ready, `glUnmapBuffer` is called to release the address mapping, and then the data are asynchronously copied to the graphics buffer by the GPU's hardware *copy engine* (a DMA device), which turns out to be more flexible and efficient.

Type-3: Drawing. After the contexts and resources are prepared, the drawing phase is usually realized with just a few API calls, for example, `glDrawArrays` as shown in Figure 2(b). Such APIs are all designed to be asynchronous in the first place, so that the graphics processing throughput of a hardware GPU can be maximized. When a drawing call is issued, the call is simply pushed into the GPU's command queue rather than being executed synchronously.

Apart from the above operations for rendering a single frame, graphics apps often need to render continuous frames (i.e., animations) in practice. To this end, a modern graphics app usually follows the *delta timing* principle [18] of graphics programming, where the app measures the rendering time of the current frame (referred to as the frame's delta time) to decide which scene should be rendered next. For example, when a game app renders the movement of a game character, the app would measure the delta time of the current frame to compute how far the character should move (i.e., the character's coordinate change) in the next frame based on the delta time and the character's moving speed.

Graphics APIs beyond OpenGL. While the above descriptions focus on OpenGL (ES), we find that the API semantics of other existing graphics frameworks (such as Vulkan) have similar characteristics. Their APIs can also be categorized into the aforementioned three types. For example, in Vulkan `VKInstance` is used for managing context information, `vkCreateBuffer` is called for allocating buffer resources, and `vkCmdDraw` issues drawing commands.

This is not surprising, but stems from a common GPU's internal design. Like a CPU, a GPU usually leverages dedicated *state registers* for determining the current operation targets and parameters (i.e., contexts), based on which an array of *computation cores* perform rendering and computing tasks in parallel. Special high-bandwidth *graphics memory* is often embedded in a GPU for holding a large amount of graphics resources (e.g., vertex and texture), therefore mitigating the *memory wall* issue observed in a CPU [77], that is, the speed disparity between memory accesses and computations. Correspondingly, the three types of graphics API calls are then used for manipulating these essential hardware components throughout a rendering thread's lifecycle.

2.2 Real-World Graphics Workloads

To obtain a deeper understanding of modern graphics workloads in terms of both control flow and data flow, we measure the top-100 3D apps (which are all game apps) from Google Play as of 05/14/2023 [75] by examining the distributions of their API calls and the sizes of their generated graphics data. We instrument vanilla Android 11's system graphics library to log the API calls and count the graphics data of a test app during its run time. For each game app, we play a full game set (whose specific operations depend on the app's content) to record the runtime API invocation data. The experiments are conducted on a (middle-end) Google Pixel 5a device, which is equipped with a Qualcomm Snapdragon 765G SoC, 6 GB memory, 128 GB storage, and 1080p display.

Figure 3 shows that an average of 2,187 API calls are issued for rendering a single frame. For most (88%) of the frames, the number of API calls is larger than 1,000. Figure 4 depicts the percentages of specific types of API calls. As shown, the distribution is quite skewed—Type-1 and Type-2 occupy the vast majority (around 94% on average), while Type-3 take up merely 6% on average. Additionally, we find that despite being the majority, most Type-1 and Type-2 calls do not have immediate effects on the final rendering results until Type-3 calls are issued. For example, graphics data stored in a graphics buffer are usually not used by the GPU before certain drawing calls are issued.

With respect to data flow, there also exists considerable disparity in the graphics data amount generated per second, as indicated in Figure 5. While 90% of the graphics data generated per second are less than 60 MB in size, the peak data rate can be as high as 1.06 GB/second, revealing significant data rate dynamics in real-world graphics workloads.

2.3 Implications for Mobile Emulation

Type-1 and Type-2 calls are relatively cheap when executed natively, but this may not be the case in a virtualized environment. If a Type-1 or Type-2 call is synchronously executed on the host GPU,

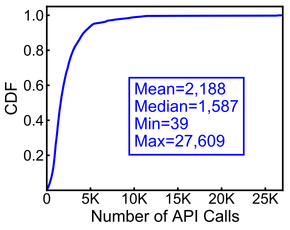


Fig. 3. Number of API calls issued for rendering a single frame.

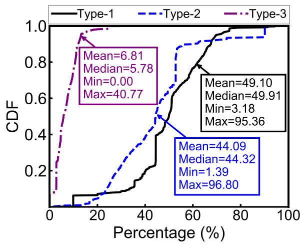


Fig. 4. Percentages of different API calls for the top-100 3D apps.

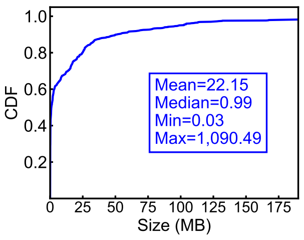


Fig. 5. Graphics data generated per second by top-100 3D apps.

it can be expensive to first exit the guest, then wait for the host to execute the call, and then return back to the guest. This "tromboning" process adds substantial latency to what might otherwise be an inexpensive call, especially when Type-1 and Type-2 calls are very frequent.

To mitigate the problem, an intuitive approach is using a buffer to batch void API calls, that is, calls that do not return any values, so that not only the void Type-1 and Type-2 calls are delayed, but the asynchronous nature of Type-3 calls (which are all void calls) can also be exploited. However, the resulting efficiency improvement is limited by the proportion of void API calls, that is, only 41.4% according to our measurement. Thus, it is no wonder that GAE, which takes this approach to improve efficiency, cannot smoothly run many common apps.

In hopes of fundamentally addressing the problem, we make the following key observation—resource-related operations (involving all Type-2 and most Type-1 operations) are fully *handle-based*. That is to say, these operations only interact with indirect, lightweight resource handles in the main memory, rather than the actual resources lying in the GPU's graphics memory. As demonstrated in Figure 2, a resource handle is merely an unsigned integer. In hardware GPU environments, this greatly facilitates the manipulation of graphics resources (without actually holding them in the main memory), thus avoiding frequently exchanging a large volume of graphics data between the main memory and the graphics memory. Note that the two memories are isolated hardware components connected via a relatively slow PCI bus.

We can exploit this key insight to accelerate mobile emulation, given that guest and host are also isolated by virtualization. We "project" a selective subset of contexts and resource handles, which are necessary for realizing actual rendering at the host GPU, onto the address spaces of guest processes; the resulting contexts after projection are termed *shadow contexts*. With the help of shadow contexts and resource handles, most (void and non-void) APIs can be asynchronously executed at the host. Moreover, certain Type-1 and Type-2 API calls (mostly used for querying context and resource information) can be directly accomplished within the projection space, *completely* eliminating their execution at the host.

3 SYSTEM OVERVIEW

Figure 6 depicts Trinity's system architecture. It uses virtualization to isolate guest and host execution environments to retain strong compatibility and security. At the heart of Trinity lies a small-size *graphics projection space*, which is allocated inside the memory of a guest app/system process. Within the space, we maintain a special set of shadow contexts and resource handles which correspond to a subset of control contexts and resources inside a hardware GPU (cf. Section 4).

Once Type-1 or Type-2 API calls issued from a guest process are executed in the projection space, the shadow contexts and resource handles will reflect and preserve their effects. Control flow then returns to the guest process for executing its next program logic without synchronously

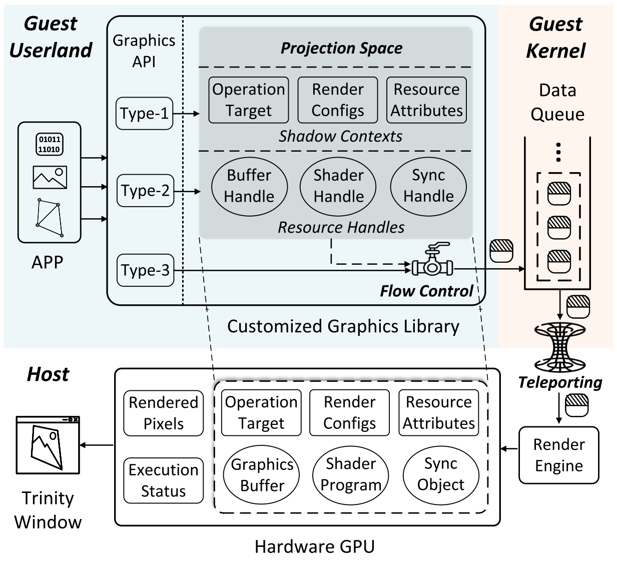


Fig. 6. Architectural overview of Trinity.

waiting for host-side execution of the API calls (as conducted by API remoting). Meanwhile, the host contexts are asynchronously aligned with the shadow contexts; mappings are asynchronously established between resource handles and host resources.

Since synchronous host-side API execution is avoided, rather than exiting to the host to deliver data, the host can choose to asynchronously fetch the guest data required for API execution from the guest memory space through polling (cf. Section 6.0.1), thus reducing frequent VM Exits. Later when the guest process issues Type-3 API calls, they are also asynchronously executed at the host as they are designed to be asynchronous. In this manner, the originally time-consuming guest-host interactions can be effectively decomposed into interleaved and mostly asynchronous guest-projection interactions and projection-host interactions.

For example, when running the program in Figure 2, Trinity directly generates a buffer handle upon the Type-2 API call `glGenBuffers`, which is then sent to the host. When the program finishes sending the handle, its control flow continues; meanwhile, the host asynchronously allocates a buffer and its handle by also calling `glGenBuffers` in a dedicated host rendering thread using the host-side desktop OpenGL library, whose APIs are a superset of OpenGL ES APIs.

The relation between the host handle and the guest one is recorded in a hash table at the host. When `glBindBuffer` (Type-1) is called with the guest handle, Trinity adjusts the shadow context information of the currently bound buffer handle, and then sends the bound guest handle to the host. When the guest finishes sending the handle, the host asynchronously looks up the corresponding host handle in the hash table, and then calls `glBindBuffer` at the host to bind the host buffer (handle) in the rendering thread.

When `glMapBufferRange` (Type-2) is called, Trinity allocates a guest memory space and returns it to the guest program. When `glUnmapBuffer` (Type-2) is called, Trinity transfers the data in the guest memory space to the host, as no further modifications can be made to the data then. At the host side, the real buffer is then asynchronously populated with the data also through `glMapBufferRange`. Finally, upon `glDrawArrays` (Type-3), Trinity asynchronously executes it at the host rendering thread, so as to instruct the host GPU to realize actual rendering with the graphics buffer's data.

To sum up, Trinity's projection space provides two key advantages. First, it helps to avoid synchronous host-side execution of APIs (as in API remoting), even for non-void calls (such as `glGenBuffers`) that need to be processed immediately, so that expensive VM Exits can also be reduced. Second, it can resolve the API calls for querying context and resource information, such as `glGetBufferParameteriv` in Figure 2, without sending them to the host. Quantitatively, 99.93% calls do not need synchronous host-side API execution, among which 26% are directly resolved at the guest (cf. Section 9.3). Although the projection space can involve processing certain calls twice—once at the guest and once at the host, this is done with relatively cheap operations whose extra costs are more than outweighed by the savings from reduced synchronous host-side execution of the APIs and the accompanied VM Exits.

To maximize Trinity's graphics processing throughput, all the above guest-side and host-side operations are coordinated by an elastic flow control algorithm (Section 5). Furthermore, the projection-host interactions are accomplished via a data teleporting method (Section 6) that attempts to maximize the data delivery throughput under high data and system dynamics.

4 GRAPHICS PROJECTION

We present the construction and maintenance of shadow contexts (Section 4.0.1) and resource handles (Section 4.0.2), that is, the key data structures that format the projection space.

4.0.1 Shadow Context. In Section 2.1, we have introduced that Type-1 APIs are usually used to manipulate three types of context information: 1) operation target, 2) render configurations, and 3) resource attributes. Apart from the above, as shown in Figure 6, context information in a real GPU environment also includes 4) rendered pixels and 5) execution status. Here, rendered pixels refer to the rendered pixels stored in graphics memory, and execution status is the current status of the GPU's command queues.

For a shadow context, we carefully select to maintain the following three types of context information: 1) operation target, 2) render configurations, and 3) resource attributes. Consequently, with the above information, subsequent reads of context information can be directly fulfilled with the shadow contexts without resorting to the host. The shadow context is maintained based on Type-1 calls issued by a guest process. For example, when the process calls `glBindBuffer` (as shown in Figure 2) to bind a buffer handle (`vertex_buffer_handle`) as the current operation target, the operation target maintained in the shadow context (usually an integer) will be modified to the buffer handle.

The other two pieces of context information we choose not to maintain, that is, rendered pixels and execution status, are related to a hardware GPU's internal states. Managing such information requires frequent interactions with the host GPU, thus incurring prohibitively high overhead. If such information is actually required, it will be retrieved from the host synchronously. Fortunately, such cases occur with a pretty low (0.07% on average) probability during an app's rendering (according to our measurement in Section 2.2). Even when such cases occur, we make considerable efforts to minimize the incurred time overhead by carefully designing the data teleporting method, which will be detailed in Section 6.

Similar to a CPU context, a rendering context is tightly coupled with the thread model of an OS. At any given point of time, a thread is bound to a single rendering context, while a rendering context can be shared among multiple rendering threads of a process to realize cooperative rendering. Thus, in the graphics projection space of a process, we maintained shadow contexts on a per-thread basis, while keeping a reference to the possible shared contexts.

4.0.2 Resource Handle. As introduced in Section 2.1, resources involved in graphics rendering include *graphics buffers*, *shader programs* and *sync objects*. Compared to contexts, the allocation of resource handles and management of actual resources often require more judicious data structure and algorithm design, as well as guest-host cooperation, since they can easily induce inefficient memory usage and implicit synchronization, thus impairing system performance.

Handle Allocation. As mentioned before, all the graphics resources are managed through resource *handles* by modern GPUs. Guided by this, when a guest process requests for a resource allocation, we directly return a handle generated by us, which is not backed with a real host GPU resource upon handle generation. Then, after the control flow is returned to the guest process, the host will perform actual resource allocation in a transparent and asynchronous manner, and record the mapping between the guest handle and the host one in a host-side hash table. To make the guest-side handle allocation efficient, we adopt a bitmap for managing each type of resource handle, with which all the resource creation and deletion can be done in $O(1)$ time complexity, and we can maintain good memory density through handle recycling.

Resource Management. After allocating resource handles for a guest process, we also need to properly manage the actual resources underlying the allocated handles. In particular, the management of buffer resources is critical to system performance as they hold most of the graphics data. As discussed in Section 2.1, there are two approaches to populating a graphics buffer with data, that is, immediate copy and latent mapping.

For the former, developers would call `glBufferData` and pass the data's memory address to the API to initiate copying the data from the main memory to the graphics buffer. In this case, we need to immediately transfer the data (upon the API call) to the host as required by the API. For the latter, as discussed in Section 3, the data transfer is conducted when the guest memory space is unmapped (i.e., `glUnmapBuffer` is called) by the guest process. When the data are transferred to the host, we need to populate the actual host-side graphics buffer with the data. To this end, we first ensure that the host context is aligned with the shadow context so that the correct buffer is bound and populated. Then, to efficiently populate the buffer, we copy the data to a graphics memory pool we maintain at the host, which maps a pre-allocated graphics memory space to a host main memory address also using latent mapping. In this way, modern GPUs' DMA copy engine can still be fully utilized to conduct asynchronous graphics buffer population without incurring implicit synchronization (Section 2.1). After the above operations are completed, the allocated guest memory space will be released, avoiding redundant memory usages.

5 FLOW CONTROL

With the guest and host control flows becoming mostly decoupled with the help of the projection space, their execution speeds also become highly uncoordinated. This is because a guest process' operations at the projection space usually only involve lightweight adjustments to the shadow contexts and resource handles, thus being much faster than host-side operations (i.e., actual rendering using the hardware GPU).

At first glance, this should not raise any problems since guest API calls that require (synchronous or asynchronous) host-side executions can simply queue up at a guest blocking queue—if the queue is filled up, the guest process would block until the host render engine finishes prior operations. However, we find that in practice this could easily lead to *control flow oscillation*. From the guest process' perspective, a large amount of API calls are first quickly handled by the projection space when the data queue is not full. Soon, when the queue is filled up, a subsequent call would suddenly take a significantly longer time to complete as the queue is waiting for the (slower) host-side actual rendering. The long processing time further leads to a long delta time of the current frame as discussed in Section 2.1. As a result, the guest process may generate abnormal animations following the delta timing principle, for example, a game character could move an abnormally long distance in just one frame due to the long delta time, leading to poor user-perceived smoothness.

To resolve this problem, instead of solely relying on a blocking queue, we orchestrate the execution speeds of control flows at both the guest and host sides. Our objective is the *fast reconciliation* of the guest-side and host-side control flows, so that the overall performance of Trinity can be staying at a high level. To this end, we design an elastic flow control algorithm based on the classic MIMD (multiplicative-increase/multiplicative-decrease) algorithm [42] in the computer networking area, which promises fast reconciliation of two network flows. To adapt MIMD to our graphics rendering scenario, we regulate control flows' execution speeds at the fine granularity of each rendered frame.

In detail, when a guest rendering thread finishes all the graphics operations related to a frame's rendering, we let it sleep for T_s milliseconds and wait for the host GPU to finish the actual rendering. T_s is then calculated as $T_s = \frac{N'}{N} \times |\overline{T_h} - \overline{T_g}|$, where N' is the current difference in the number of rendered frames between the guest's and host's rendering threads, N is the desirable maximum difference set by us (N is currently set to 3 in Trinity as we use the widely-adopted *triple buffering* mechanism for smooth rendering at the host), $\overline{T_h}$ is the host's average frame time (for executing all the graphics operations related to a frame) for the nearest N frames, and $\overline{T_g}$ is the guest's average frame time also for the nearest N frames. $\overline{T_h}$ and $\overline{T_g}$ are calculated by counting each frame's rendering time at the host and the guest sides.

Specifically, if $N' > N$ (i.e., the guest is too fast), T_s will be multiplicatively increased to a longer time to approximate the host's rendering speed. Otherwise, T_s will be multiplicatively decreased, striving to maintain the current frame number difference at the desirable value. Typically, T_s lies between several milliseconds and tens of milliseconds depending on the guest-host rendering speed gap. In this way, Trinity can quickly reconcile the guest-side and host-side control flows.

6 DATA TELEPORTING

Fast guest-host data delivery is critical for keeping projection-host interactions efficient. To realize this, we first analyze system and data dynamics (Section 6.0.1) that constitute a major obstacle to the goal, and then describe the workflow of our data teleporting method (Section 6.0.2), which leverages static timing analysis to accommodate the dynamic situations.

6.0.1 System and Data Dynamics. When control flows are synchronously accompanied by data flows, the guest-host data delivery mechanism can be very simple. For example, in API remoting, VM Exits/Enters are leveraged to achieve control handover and data exchange at the same time. In Trinity, however, data flows are decoupled from control flows (thanks to the graphics projection space), so we are confronted with complicated situations as well as design choices. Among these data flows, projection-host data exchanges are the most likely to become a performance bottleneck due to their crossing the virtualization boundary.

By carefully analyzing the projection-host data exchanges when running top-100 3D apps, we find that the major challenge of rapidly delivering them lies in the high dynamics of system status and data volume (abbreviated as *system dynamics* and *data dynamics* respectively). With regard to system dynamics, the major impact factors are the available memory bandwidth and current CPU utilizations, which are not hard to understand. As to data dynamics, call data of APIs that require synchronous host execution are sensitive to end-to-end latency (i.e., the delay until host-side executions of the calls), while asynchronous ones require high processing throughput. Further, we pay special attention to distinct data sizes and bursty data exchanges (i.e., bulk data exchange during a short period of time) which are common in modern graphics workloads as shown in Figure 5. In general, we can classify the dynamic situations into ~16 patterns, roughly corresponding to the combinations of 1) high/low CPU utilization, 2) high/low available memory bandwidth, 3) synchronous/asynchronous API call data, and 4) large/small data sizes.

To accommodate the dynamic situations, our key observation is that the guest-host data delivery process can be decomposed into three stages, that is, *data aggregation*, *data persistence,* and *arrival notification*, as the data travel through the guest user space, the guest kernel space and the host. Moreover, in each of the stages, we find that there are mainly two different data delivery strategies, which make opposing tradeoffs under different dynamic situations as discussed below.

— Data Aggregation. As exercised in GAE, aggregating non-void API calls with a user-space buffer can usually reduce the frequency of user/kernel switches. This is also the case for Trinity since host-side execution of API calls is mostly asynchronous. However, if the data to be transferred are particularly large (e.g., in bursty data exchanges), memory copies during data aggregation could bring larger time overhead compared to user/kernel switches; hence, the data should be delivered to the kernel as early as possible without any aggregation.
— Data Persistence. For the data of a guest rendering thread, we need to ensure their persistence until they are fetched by the host. To this end, a simple strategy is blocking the thread's control flow until the data delivery is done (as adopted by GAE). In Trinity, we realize that there is an alternative strategy by using a special persistent space (e.g., in the guest kernel) to maintain the guest thread's data, so that there is no need to block the thread's control flow. Intuitively, this strategy is most suited to small data delivery, which does not incur long-time memory copies.
— Arrival Notification. To notify the host to fetch the data that have arrived, we can simply leverage the VM Exit-based strategy (adopted by GAE), whose incurred delays can be as low as tens of microseconds. This, however, can lead to the guest core's being completely stopped. Alternatively, for asynchronous data fetching, we can utilize a data polling-based strategy at the host, which does not incur the guest world's stopping but would introduce millisecond-level delay due to the thread sleeping and CPU scheduling delays of a common time-sharing host OS.

6.0.2 Workflow. Given that there is no single strategy that can accommodate every dynamic situation, we implement in Trinity all the combinations of strategies. Almost all of them are implemented at the guest side, except that data polling is realized by the host.

To decide the proper strategy during each stage of data delivery, we adopt the *static timing analysis* [13] method, which calculates the expected delay of each *timing step* (i.e., stage) incurred by different data delivery strategies. As mentioned before, the stages include data aggregation, data persistence, and arrival notification. Suppose a guest app wishes to deliver a data chunk of size S_{data}, the current copy speed of the guest memory is V_{guest}, the current copy speed of the host memory is V_{host}. Below we elaborate on the workflow of data teleporting which selects the suitable strategy in each data delivery stage based on static timing analysis.

Data Aggregation. As shown in Figure 7, if the data to be delivered are asynchronous API call data (i.e., call data of APIs that do not need synchronous host-side execution), we can aggregate them in a user-space buffer to reduce projection-host interactions. However, aggregating the data in the buffer incurs a memory copy, resulting in a delay of $\frac{S_{data}}{V_{guest}}$. Otherwise, an individual `write` system call will be invoked to write the data to our kernel character device driver (cf. Section 8), whose time overhead is T_{write}. Obviously, if $\frac{S_{data}}{V_{guest}} < T_{write}$, we choose to aggregate the data; else, we choose not to.

In contrast, for synchronous API call data we should always avoid data aggregation since synchronous calls should be immediately delivered to the host for executions. Then, along with these

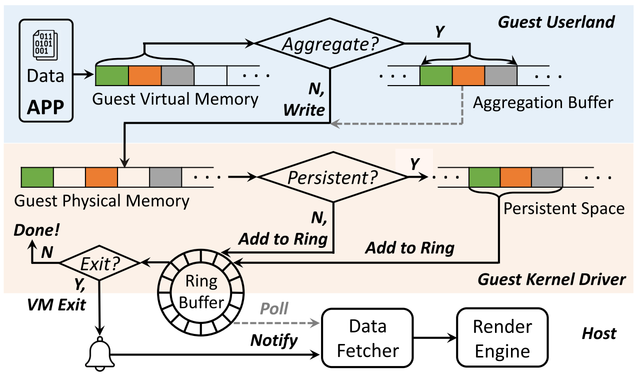


Fig. 7. Workflow of data teleporting.

non-aggregation data, the aggregation buffer will also be written to our kernel driver and then cleared. We next enter the data persistence stage.

Data Persistence. In this stage, our kernel driver will decide whether to block the guest app's control flow, or utilize an additional persistent space for ensuring the persistence of a guest thread's data until the data are fetched by the host. Unlike the user-space data aggregation buffer that serves to reduce the frequency of entering the kernel and interacting with the host, the kernel persistent space allows the app's control flow to quickly return to the user space for executing its next logic. In practice, if we resort to the control flow blocking strategy, the blocking time will consist of four parts: 1) the delay of adding the data to a ring buffer shared by the guest and the host for realizing data delivery—T_{ring}, 2) the delay of host notification—T_{hn}, 3) the time for a host-side memory copy to fetch data (detailed later in Data Fetching)—$\frac{S_{data}}{V_{host}}$, and 4) the delay of host-to-guest notification through interrupt injection for returning the control flow to the guest app—T_{gn}. Here, the ring buffer does not directly store the data; instead, to transfer a large volume of data, it holds a number of (currently 1,024) pointers, each of which points to another ring buffer of the same size, whose buffer item stores the data's physical addresses. Therefore, the blocking strategy's time overhead $T_{blocking}$ is the sum of them: $T_{blocking} = T_{ring} + T_{hn} + \frac{S_{data}}{V_{host}} + T_{gn}$. Here, we encounter a challenge: T_{hn} is dependent on the arrival notification strategy, which we have not decided yet. Fortunately, we find that when the control flow blocking strategy is adopted, the app thread's execution flow has already stopped. Thus, a VM Exit's side effect no longer matters in this case, but its advantage of short delay makes it an appropriate choice. We then naturally take the VM Exit-based arrival notification strategy, so T_{hn} generally equals the delay of a VM Exit.

On the other hand, if we choose to leverage a kernel persistent space for data persistence, the time overhead comes from 1) a memory copy to the persistent space and 2) adding the data to the ring buffer, that is, $T_{persistent} = \frac{S_{data}}{V_{guest}} + T_{ring}$. After the above are finished, the guest app's control flow is immediately returned to its user space for executing its next program logic, while the host asynchronously polls for data arrival and fetches data (as to be detailed later).

Based on the calculated $T_{blocking}$ and $T_{persistent}$, we can then choose the data persistence strategy with a smaller delay. Also, for synchronous API call data, we directly choose the blocking strategy because during synchronous calls the control flow is naturally blocked until host-side executions. With respect to the parameters used in the above analysis, they can be either directly obtained (e.g., S_{data}) or statistically estimated by monitoring their recent values and calculating the average (e.g., V_{guest} and V_{host}).

Arrival Notification. After the data are added to the ring buffer, we then need to choose a proper strategy for notifying the host of data arrival. In practice, we find that the arrival notification strategy is closely related to the data persistence strategy. Specifically, control flow blocking is particularly sensitive to the arrival notification delay, and thus should be coupled with VM Exits. On the contrary, the persistent space-based strategy allows arrival notification and data fetching to be asynchronous, and thus the polling-based strategy should be selected; the polling is performed by a host-side data fetching thread (referred to as Data Fetcher) every millisecond.

Data Fetching. When Data Fetcher is notified of data arrival, it would read the ring buffer to acquire the data. If the data are contiguous in the guest physical memory (and thus contiguous in the host virtual memory), the data can be directly accessed without further memory copy; otherwise, they should be copied to a contiguous host buffer. The fetched data are then distributed to the host render engine's rendering threads for realizing actual rendering.

7 RELIABILITY ISSUE ANALYSIS PIPELINE

While improving performance, the decoupled and asynchronous nature of graphics projection has significantly complicated the control flows of Trinity, making it easily subject to considerable reliability issues, especially when facing a variety of uncommon or non-standard app behaviors in the wild. To facilitate the deployment and maintenance of Trinity in production, in this section, we present the design of an automatic diagnosis pipeline in Trinity for complex reliability issues.

7.1 Understanding Cross-Layer Reliability Issues

With the introduction of graphics projection, we find that in practice it is hard to diagnose and address reliability issues such as functional failures, app crashes, and system hangs. In most cases, the direct failure sites are both spatially and temporarily distant from the root causes, for example, failures occurring at the guest side may stem from an incorrect state of the host (who does the actual rendering work) and may only manifest when the faulty state asynchronously propagates to the guest at a later time, due to the decoupled control and data flows of the projection space. As a result, the symptoms and the event call stacks of such an issue are often distinct and hard to correlate at the guest and the host sides. In this work, we refer to these issues as *cross-layer reliability issues*.

Figure 8 demonstrates a typical cross-layer reliability issue in Trinity. As shown, due to the architectural differences between the mobile GPU and PC/server GPU, mobile-specific features supported by the mobile graphics shader languages (used by the guest app) may be missing in the PC/server graphics shader languages (used by Trinity's host-side render engine), leading to compilation errors when the host attempts to compile the shader sources of the guest app. Such errors will not immediately result in failures at the host side, and thus the host will continue to establish a mapping between the shader handle produced by the projection space and the erroneous host shader (as described in Section 4). However, when the guest app invokes shader APIs dependent on mobile-specific features via the projection space, the errors will finally manifest themselves as failures at the guest side, significantly misleading debugging and diagnosis efforts. The proprietary and closed-source nature of most commercial apps further hinders problem analysis.

7.2 Design Overview

To address the above problem, we develop an automatic root cause analysis pipeline for diagnosing cross-layer reliability issues in Trinity. Our pipeline not only helps considerably boost Trinity's reliability in operational environments, but also provides key experiences for diagnosing and

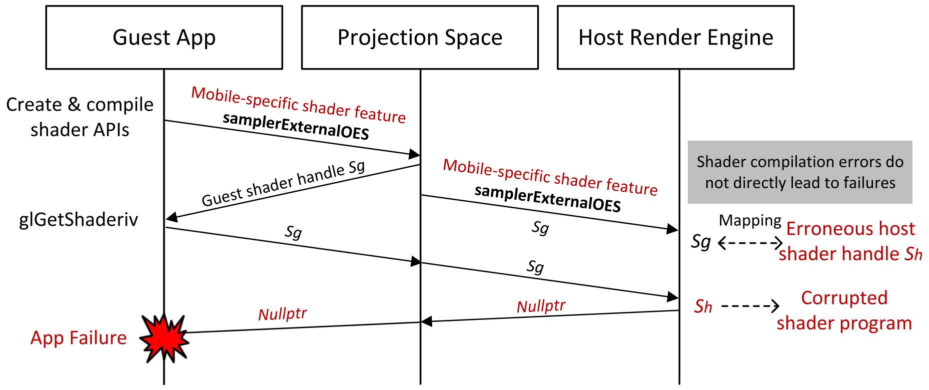


Fig. 8. A typical cross-layer reliability issue in Trinity.

debugging modern virtualization systems, where cross-layer design (e.g., paravirtualization-based device emulation) is widely adopted to optimize system efficiency.

Our key insight regarding cross-layer reliability issues is that they stem from *state discrepancies* between the guest and the host, which are context and resource value differences caused by the different behaviors and the same API at the guest and host sides. For example, the failure shown in Figure 8 roots in the handle discrepancy of shader resources caused by different shader creation and compilation API behaviors at the guest and host. The latent discrepancies can be (sometimes randomly) triggered by the guest or the host, resulting in temporal and spatial displacements of the failures from their root causes. Based on the insight, our idea of diagnosing reliability issues is to introspect and differentiate the states of the guest-side and host-side contexts and resources, so that we can detect discrepancies related to the issue in a timely and local fashion.

Realizing this in our cross-layer graphics virtualization system cannot be easily achieved with existing diagnosis tools. For instance, a variety of **virtual machine introspection** (**VMI**) systems have been developed to probe the internal states of a VM from the host at run time. VMI systems typically record guest OSes' low-level status such as CPU instructions [78], system calls [19], I/O activities [60], and memory data [65] to identify problematic control and data flows. However, it is difficult to reconstruct high-level guest application states from the low-level instructions/memory bytes. To overcome the semantic gap, prior work mainly focuses on the specific characteristics of a target runtime, for example, DroidScope [78] analyzes the Android Java VM's memory and instruction characteristics to correlate them with low-level information, while Virtuoso [19] learns the correlation by tracing the CPU instructions produced by the system calls/APIs of a target OS. Unfortunately, we are not aware of any VMI methods that are able to bridge the gap for the mobile graphics system. Also, porting existing VMI methods is challenging because various rendering states are not present in the CPU or memory controlled by the VM but live within the GPU, and thus are hard to directly introspect. Further, VMI methods require frequently pausing the guest VM to achieve introspection, raising performance concerns.

There also exist several tools that perform in-app method tracing, such as Android Systrace [7], Firebase [36], AppInsight [67], and Hubble [53]. In practice, they are useful for debugging app-level performance issues by logging the method/API execution time and identifying the critical path. In theory, they can also be adapted to log all calls to APIs that manipulate state and data in the projection space to realize state introspection. However, we find that the static logs can be confusing when multiple state differences are recorded during apps' execution (which is common in practice), and thus still require nontrivial manual analysis efforts.

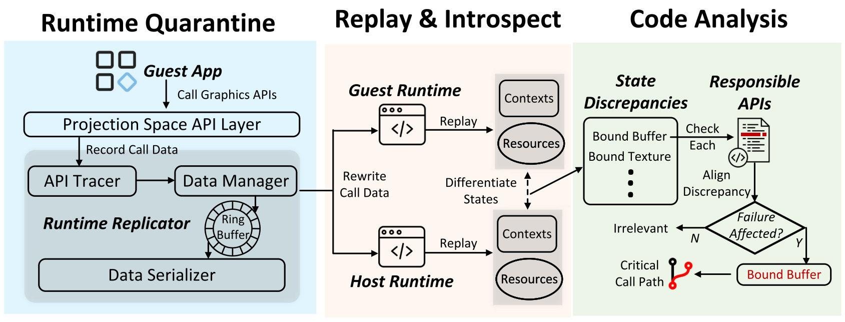


Fig. 9. Workflow of our root cause analysis pipeline.

A plausible alternative is deterministic **record and replay** (**R and R**) [17, 23, 24, 47, 48, 59], which can fully capture the failure scene by logging nondeterministic events like user inputs, shared memory accesses, scheduling events, and so on. at run time. R and R enables more insightful debugging practices like cyclic and speculative debugging. However, similar to VMI, prior R and R methods focus on the low-level instruction/data flows that are hard to correlate with high-level rendering states. Also, their generic designs can incur high overhead if directly applied to graphics systems (e.g., incurring large data copies), due to their unique control and data characteristics (Section 6.0.1).

To account for these challenges, we develop an efficient runtime quarantine technique which replicates and isolates the control and data flows of the guest and the host. The quarantined control and data flows then intrinsically contain the information and changes of all rendering states, therefore enabling us to achieve state introspection later in an offline manner without modifying or heavily impacting the critical code path of the guest and the host. The high-level goal of our method is similar to R and R but we address critical challenges in virtualized graphics systems based on the unique opportunity of mobile graphics frameworks.

Figure 9 illustrates the workflow of our root cause analysis method, which works in a three-stage fashion. First in the runtime quarantine stage, the target guest app is launched while the guest-side runtime replicator begins recording its API usages and resource data. Upon the occurrences of failures, the replicator would stop and output the replicated runtime as standalone, executable programs, which are leveraged to reproduce the failures and facilitate diagnosis later. With this, we then enter the replay and introspect stage where we execute the replicated runtime natively and outside of the emulation environments. During replaying, we locate any state discrepancies occurring before the failures by differentiating the values of key context and resource states (i.e., those managed by the projection space) between the guest-side and host-side runtime. Finally, in the code analysis stage, we distill the critical code paths (that essentially lead to reliability issues) by perturbing the parameters of the API calls related to the states and checking whether the failure scene is affected in the meantime. We next describe the specific designs of each stage.

7.3 Runtime Quarantine

To realize runtime quarantine, we exploit the projection space's role as the graphics API provider of the guest OS, which sits between guest apps and the underlying emulator and handles all guest apps' graphics API invocations and graphics resources. Based on this special opportunity, our basic idea is to perform in-memory tracing of a guest app's graphics API invocations (i.e., control flows, including the invocation timestamp) and the data passed to the APIs (i.e., data flows) during the app's execution. As the API provider, we do not require invasive and inefficient API hooking techniques used by commercial debugging tools for graphics rendering (e.g., RenderDoc [69]) to realize this, as well as any knowledge of guest apps' source code. Also, it avoids retrievals or modifications of any context/resource states, thereby faithfully preserving the failure scenes. Most importantly, the quarantined control/data flows enable us to perform more informed and active runtime behavioral analysis of the code, as to be detailed in Section 7.4.

Runtime Replicator Design. Nevertheless, blindly recording all API invocations and involved data can easily incur non-trivial overhead. Our experiments (detailed hardware/software configurations are listed in Section 9.1) also show that preserving all the API traces and the corresponding data consumes up to ~1 GB memory and ~50% CPU for a heavy 3D app, which can severely impact the performance and hinders the debugging of failures that require a long execution time to produce. In particular, we find that over 95% of the consumed memory and CPU are leveraged to store and process the large volumes of resource data of 3D apps.

To effectively reduce the overhead, we make three-fold efforts in both the design and implementation of our runtime replicator by taking advantage of the characteristics of mobile graphics APIs (Section 2), especially those related to resource data management. First, we design the replication of resource data to be lazy and intelligent, that is, the resource data uploaded (through resource management APIs) by a guest app are only replicated and serialized by us when they are used by the app. Also, we identify resource data shared by multiple rendering contexts (by comparing their memory addresses) to avoid redundant replications in practice. These are based on our finding that only a small portion (~10% on average according to our measurements) of resource data uploaded to the GPU memory by an app are actually used during the rendering of a specific scene, because modern 3D apps tend to actively use available memory space for caching large resource data (e.g., textures and vertexes) upon launch time to maximize their performance during rendering.

Second, we recognize context/resource creation and deletion API calls to further reduce overhead. In graphics APIs, when a rendering context (or resource) is unbound and destroyed, its related API calls and resources become obsolete and thus are safe to discard to save memory and computation, unless the context is previously shared by other contexts. This often occurs during rendering scene transitions, for example, from the welcome screen of a 3D game to the actual game scene. Therefore, by safely removing destroyed contexts and resource data, we can avoid the excessive overhead for recording rendering operations that are not related to the problem.

Third, we build the runtime replicator atop a special lock-free ring buffer that allows simultaneous reads and writes without synchronizations, enabling asynchronous compression of the replicated data and background flushing to the disk. Similar to typical lock-free data structures used by Linux kernel [51] and other highly-parallel software like DPDK [21], our ring buffer utilizes atomic CPU instructions to achieve concurrent reads and writes. During an app's execution, tracing data are written into the ring buffer and the data's tail is referenced by a pointer. In the meantime, a background serialization thread periodically reads the data from the buffer and flushes them to the disk in a frame-by-frame manner (i.e., the tracing data of a frame is atomically read out and flushed) to ensure the data integrity of a whole rendering frame; the read position is also referenced by a pointer to realize the lock-free mechanism. Upon serialization, texture data (which are generally image data) are encoded into the PNG format to save space, while vertex data (which are numeric array data) are compressed with the state-of-the-art Zstandard algorithm [26].

Assembling the above designs, we can efficiently realize runtime quarantine for the control and data flows of guest apps. To complete the design, we also need to quarantine the corresponding host-side control/data flows to achieve state introspection and differentiation. Fortunately, with the guest-side runtime replicator, this is straightforward to implement as we only need to capture the related projection-host interactions (which are mostly remote procedural calls from the projection space to the host) when a graphics API is invoked. For the data flows, resource data normally do not require additional replications as they are the same as those of the guest-side data flows, except when data are corrupted due to memory overflow or incorrect data encoding in the projection space. We can identify such corruptions through the lightweight hashing of the data.

Runtime Replicator Implementation. We implement the runtime replicator into three major components as shown in Figure 9 in the projection space, that is, API tracer, runtime data manager and data serializer. API tracer directly interfaces with the graphics API layer of the projection space by automatically instrumenting the APIs. When a guest app invokes a graphics API to conduct rendering, API tracer passes the API's ID (i.e., a unique unsigned integer assigned to each API by us), the invocation timestamp, its parameter data, and the corresponding projection-host interactions to the runtime data manager. Based on our designs described above, the runtime data manager then decides how to handle the data, for example, whether the data should be lazily replicated; if the data should be replicated, it then copies the data to the lock-free ring buffer for in-memory preservation. Meanwhile, the data serializer, which runs in a separate thread, periodically and asynchronously reads the buffer and writes the compressed data to disk as described above. Our implementation only requires modifications to the APIs' entry points to realize instrumentations, which are uniformly realized at API definitions, while the other components are largely decoupled from the original code base. In this way, we can easily isolate the failures in Trinity from our analysis pipeline.

Overhead Analysis. We measure the runtime overhead incurred by the runtime replicator using the top-100 3D apps from Google Play and the software/hardware setups described in Section 9. On average, our runtime replicator can achieve efficient runtime quarantine with 5% CPU and 14 MB memory overhead for the apps. The average delay to a frame's rendering (which should be 16.67 milliseconds for an app with 60 Hz refresh rate) is less than 1 millisecond, therefore does not incur noticeable performance penalty on the execution of the apps. In the worst-case scenario, we observe 12% CPU and 63 MB memory usage for a heavy 3D app. In practice, the overhead is still acceptable by causing only up to 1.5 milliseconds of delay per frame.

7.4 Introspection-Guided Code Analysis

With the efficient runtime quarantine infrastructure, we preserve the runtime of failure scenes and enable active state introspection as well as dynamic behavioral analysis off the "hot path" of the guest rendering tasks, therefore avoiding heavily interfering with the original runtime. We next describe the detailed design of our root cause analysis method based on runtime state introspection.

Quarantined Runtime Replay. Recall that state discrepancies of contexts and resources between the guest and host is the major characteristic of cross-layer reliability issues. Based on the finding, our idea is to compare the runtime states of the guest and the host, so as to pinpoint the root causes and facilitate problem resolution. To realize this, we replay the quarantined guest-side and host-side runtime (i.e., control and data flows) in native environments, that is, native machines running the guest system (Android) and the host system (Windows or macOS), respectively. In general, this is done by executing each API traced by us with the associated data. Also, we strictly ensure the original execution order of APIs (acquired via the API invocation time), so as to faithfully reproduce the failure scenes, particularly those related to data races.

However, not all APIs can be verbatim executed with the recorded parameter data, due to inevitable differences in the execution contexts between the original runtime and the quarantined runtime, for example, their memory layouts. To adapt to this, we rewrite the API data that are subject to the problem, which involve two kinds of APIs—resource creation APIs and resource upload APIs.

For the resource creation APIs, we find that the values of the resource handles generated by them can vary significantly among different executions and are mostly unpredictable, because they are decided by the underlying GPU drivers over which we have no control. As a result, when running the quarantined runtime, the generated resource handles' values may be distinct from those produced in the original runtime. Directly using the original handle values is therefore error-prone. We thus establish a mapping between the original handle values and the new values produced when executing the quarantined runtime, and rewrite all the parameter data that contain the original values to their quarantined counterpart.

For the resource upload APIs that transfer resources from the main memory to the GPU memory, their parameter data include main memory addresses pointing to the resources in the memory, whose layout, however, has completely changed when executing the quarantined runtime. To account for this, we first need to complement the runtime by adding instructions that allocate memory spaces and store the corresponding resources in them. We then rewrite the original addresses to the allocated addresses to correctly perform resource uploads to the GPU.

State Introspection and Code Analysis. Having been able to replay the quarantined runtime, we then differentiate the critical states of contexts and resources between the guest and the host, so as to reveal the state discrepancies that result in failures. To achieve this, during the execution of the quarantined runtime, we compare key data structures maintained by the guest-side projection space regarding context/resource states with those of the host GPU when running the host-side quarantined runtime, including bound buffers, resource specifications, and shader status. Note that we only perform introspections when APIs that modify the concerned states are invoked, and only the modified states and their relevant states (rather than all states) are examined.

In practice, there can exist multiple discrepancies (up to 38 in practice) upon failure occurrences, while a large portion of them are in fact irrelevant to the problem, for example, certain context/resource states may be initialized to different values at the guest and the host by their respective drivers but are later reset to be the same, and thus are not responsible for the failure. To locate the true culprit, we perform introspection-guided code analysis at the quarantined runtime. Specifically, for an uncovered state discrepancy of a failure event, we first extract the API that first modifies the state (at the guest side or the host side) and incurs the discrepancy. We then perturb the API's behavior to check whether the failure occurrence is affected (i.e., whether the failure scene is changed), so that we can infer the causality between the API/state and the failure and rule out non-essential state discrepancies.

To this end, if the concerned APIs are invoked by the guest (or the host), we can adjust the parameter data of the APIs based on the context value of the host (or the guest) to align the context states between the guest and the host; if the states are not explicitly set by the APIs (e.g., default state values), we can insert additional API calls to align the states, while carefully maintaining the execution order of the original APIs. However, in a small portion of cases such API behavior perturbations cannot successfully align the states because the discrepancies root in deeper causes which cannot be directly addressed by resetting the states, for example, the discrepancy shown in Figure 8 is incurred by platform incompatibility, which cannot be aligned by adjusting the parameters or recreating the shader program. When these cases occur, we should always pay special attention to them and determine them as possible sources of failures.

Finally, after the above analysis, we can obtain a small set of state discrepancies (usually less than 5) that are highly likely to be the causes of the failure event. Our pipeline then extracts the code paths that modify the states (which can also be obtained from the above analysis) from Trinity's code base to facilitate further manual analysis, significantly reducing the debugging space for us.

7.5 Case Study

We leverage the failure event shown in Figure 8 as an example use case of our analysis pipeline. We first execute the problematic program with the runtime replicator turned on to trace the API and data flows at the guest and host sides. After the program crashes, we perform introspection-guided code analysis which pinpoints the shader error status value discrepancy after shader compilation at the guest-side and host-side quarantined runtime, and further extracts the code path `glCreateShader`→`glShaderSource`→`glCompileShader` that is responsible for shader creation/compilation as one of the possible root causes of the failure. After that, manual analysis can easily discover the compilation errors at the host. We will demonstrate the practical effectiveness and applications of our pipeline in Section 7.6.

7.6 Analysis Effectiveness and Reliability Enhancements

In November 2022, we applied the developed analysis pipeline to all the cross-layer reliability issues captured in production testing and reveal their major root causes, for which we provide solutions that yield impacts beyond the scope of Trinity. In particular, our experiences show that the analysis pipeline can narrow down the scope of problematic code to an average of 10 lines, largely facilitating the root cause analysis and bug fixing. Most importantly, manual analysis of the problems shows that the pipeline can correctly retrieve *all* the state discrepancies and problematic code paths that result in the concerned failures, while introducing only 8% false positives, that is, discrepancies that are not related to the failures. Such false positives are mostly state discrepancies incurred by API behavioral distinctions between the guest and host GPU platforms, which do not manifest as failures in the events but can still result in other issues such as inconsistent rendering effects.

7.6.1 Root Causes. Our analysis pipeline and manual analysis pinpoint three-fold major root causes: 1) inconsistent context and resource lifecycle between the projection space and the host GPU (46%), 2) non-standard API usages by guest apps (33%), and 3) incorrect API behavior definitions in the OpenGL document (21%). A typical case of the first root cause occurs when the guest app finishes the execution of all API calls for a rendering Surface, and submits the Surface to the `SurfaceFlinger` system service in Android for compositions. Due to the asynchronous rendering of the projection space and the asynchronous relationship between the guest app and `SurfaceFlinger`, when the host begins actual compositions (upon receiving the API calls from `SurfaceFlinger`), the Surface's lifecycle may have already ended, that is, destroyed by the guest app, leading to illegal memory access at the host side. Similar problems often occur during the multi-threaded/process rendering of recent game engines (such as Unity and Unreal Engine), where contexts and resources shared by multiple rendering threads/processes can easily lead to inconsistent lifecycle at the guest and host sides, causing considerable data race and corruption.

On the other hand, the latter two root causes are surprisingly not incurred by Trinity but stem from the bugs of guest apps and the OpenGL API standard. For example, we find that a popular 3D gaming app tested by us often maps a graphics buffer to the main memory to modify the data in the buffer by calling `glMapBufferRange` with the flag set to `GL_MAP_WRITE_BIT`, meaning that the buffer is mapped for writing/modification. However, although the app only modifies a portion of the buffer data, it maps the entire buffer to the memory. This incurs undefined behaviors in OpenGL (ES) APIs, because for GPUs with dedicated memory, the original data of the buffer will not be read to the main memory when the mapping is flagged as `GL_MAP_WRITE_BIT`, so that unnecessary GPU-to-memory data transfer can be avoided. Therefore, when the entire buffer is mapped, the mapped space in fact does not contain the original buffer data. Consequently, if the guest app only writes a portion of the data to the mapped memory space, the other part of the data in the space will still be uninitialized; when the mapped memory space is again written to the GPU's graphics memory, the buffer data are then corrupted by the uninitialized data and lead to failures.

7.6.2 Reliability Enahancements. Targeting the above root causes, we develop practical enhancements to Trinity's design and beyond, successfully resolving all the uncovered reliability issues in production testing.

Lifecycle Barrier Instruction. To address the first (also the largest) root cause, our key observation is that such data races and corruptions among multiple threads mainly occur during critical lifecycle stages of the contexts and resources, for example, creation and destruction. We thus propose to extend the original OpenGL APIs by adding special "lifecycle barrier" instructions for concerned contexts and resources, which resemble the memory barrier instruction for out-of-order CPU pipelines. When the instructions are triggered, we enforce a lightweight synchronization between the projection space and the host. In this manner, we ensure that the APIs invoked before and after a critical lifecycle APIs are executed at the host in their original order, so that lifecycle misalignment can be avoided. Finally, we identify the context/resource lifecycle APIs in the projection space and insert the barrier instructions to fix the issues. Also, we find that for certain high-level OpenGL resources (like Surface), their multiple instances in fact correspond to the same low-level Android component (like GraphicBuffer). We thus identify the underlying component during the related resource creation stage to avoid excessive and duplicate synchronization.

API Behavior Alignment. For the second root cause, we deal with it by adapting Trinity's guest-side API behaviors to accommodate guest apps' non-standard API usages. For example, we adjust the processing logic for apps calling `glMapBufferRange` with the `GL_MAP_WRITE_BIT` flag by populating the mapped buffer with the original data if the mapped range does not match the write range. For the third root cause (i.e., incorrect API behavior definitions in the OpenGL document), we not only handle them by adjusting our own API implementations, but also report the problematic API definitions to Khronos, the authoritative organization that manages the OpenGL standard, to benefit other OpenGL implementations, for example, SoC vendors' GPU drivers. The bug reports [74] have been confirmed, and the suggested fixes are currently going through reviews before being merged into the official OpenGL document.

8 IMPLEMENTATION

To realize Trinity, we make multiple modifications to the guest Android system and QEMU. First, we find that Android (as well as many UI-centric systems) clearly separates its versatile user-level graphics frameworks/libraries [6, 71] from the underlying system graphics library that realizes actual rendering. This enables us to effectively delegate every graphics API call by customizing only the system graphics library. At the guest user space, we replace the original system graphics library (i.e., `libGLES`) with our customized one, which maintains the projection space and conducts flow control. The library exposes the standard OpenGL ES interfaces to apps, allowing them to seamlessly run without modifications.

To execute the delegated Type-1 and Type-2 APIs in the projection space, we implement all of them in the system graphics library, involving a total of 220 Type-1 APIs, 128 Type-2 APIs and 10 Type-3 APIs, which fully cover the standard OpenGL ES APIs from OpenGL ES 2.0 to the latest OpenGL ES 3.2. Additionally, we implement all the 54 Android Native Platform Graphics Interface (EGL) [5] functions to interface with the Android native window system. In practice, many APIs have similar functions, simplifying their implementations, for example, `glUniform` has 33 variants used for data arrays of different sizes and data types, such as `glUniform2f` for two floats and `glUniform3i` for three integers.

At the guest's kernel space and the host, we realize data teleporting via a QEMU virtual PCI device and a guest kernel driver. As a typical character device driver, our kernel driver mounts a device file in the guest filesystem, where the user-space processes can read from and write to so as to achieve generic data transferring. With this, API calls that require host-side executions are compacted in a data packet and distributed to our host-side render engine. The render engine then leverages the desktop OpenGL library to perform actual rendering using the host GPU.

Trinity is implemented on top of QEMU 5.0 in 118K lines of (C/C++) code (LoC). In total, the projection space, flow control and data teleporting involve 113K LoC, 220 LoC and 5K LoC, respectively. Among all the code, only around 2K LoC are OS-specific, involving kernel drivers and native window system interactions.

Trinity hosts the Android-x86 system (version 9.0). Since our modifications to QEMU and Android-x86 are dynamic libraries and additional virtual devices, they can be easily applied to higher-version QEMU and Android. Trinity can run on most of the mainstream OSes (e.g., Windows 10/11 and macOS 10/11/12) with both Intel and AMD x86 CPUs. It utilizes hardware-assisted technologies (e.g., Intel VT and AMD-V) for CPU/memory virtualization. For the compatibility with ARM-based apps, Trinity incorporates Intel Houdini [40] into the guest system for dynamic binary translation.

Finally, to continuously improve Trinity's reliability, we implement our runtime quarantine infrastructure of the proposed reliability issue analysis pipeline into the projection space as an on-demand component, which can be activated by developers to capture failure scenes and perform in-depth analysis in the future.

9 EVALUATION

We evaluate Trinity with regard to our goal of simultaneously maintaining high efficiency, compatibility, and reliability. First, we describe our experiment setup in Section 9.1. Next, we present the evaluation results in Section 9.2, including 1) Trinity's efficiency measurement with standard 3D graphics benchmarks, 2) Trinity's smoothness situation with the top-100 3D apps from Google Play, and 3) Trinity's compatibility and reliability on 10K apps randomly selected from Google Play. In particular, we compare the results before (still referred to as Trinity) and after (Trinity 2.0) applying our reliability enhancements. Finally, we present the performance breakdown in Section 9.3 by removing each of the three major system mechanisms—projection space, flow control and data teleporting.

9.1 Experiment Setup

To understand the performance of Trinity and Trinity 2.0 in a comprehensive manner, we compare them with six mainstream emulators, including GAE, QEMU-KVM, VMware Workstation, Bluestacks, and DAOW, as well as WSA—a Hyper-V-based emulator released in Windows 11. Their architectures and graphics stacks are shown in Table 1. We use their latest versions as of May. 2023.

Software and Hardware Configurations. Regarding the configurations of these emulators, we set up all their instances with a 4-core CPU, 4 GB RAM, 64 GB storage, and 1080p display (i.e., the display width and height are 1920 pixels and 1080 pixels, respectively) with 60 Hz refresh rate. However, since WSA does not allow customizing configurations, we use its default settings which utilize the host system's resources to the full extent. For other options (e.g., network) in the emulators, we also leave them as default.

Table 1. Comparison of the Evaluated Emulators

Mobile Emulator	System Architecture	Graphics Stack
GAE [32]	x86 Android on customized QEMU	API remoting
WSA [58]	x86 Android on Windows Hyper-V	API remoting
QEMU-KVM [66]	Android-x86 on QEMU	Device emulation
VMware Workstation [76]	Android-x86 on VMware Workstation	Device emulation
Bluestacks [15]	Android-x86 on VirtualBox	Proprietary
DAOW [80]	Direct Android emulation on Windows	API translation with ANGLE [31]

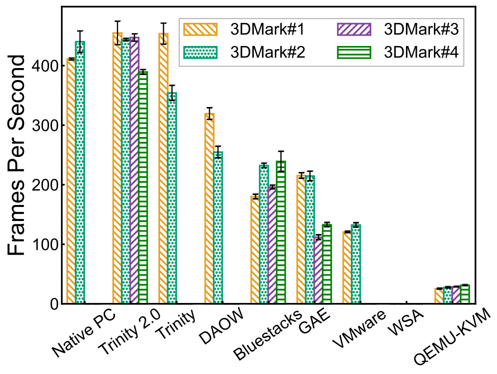
Table 2. List of the Evaluation Benchmarks

ID	Full Benchmark Name	Benchmark App
3DMark#1	Slinghshot Extreme Test 1	3DMark
3DMark#2	Slinghshot Extreme Test 2	3DMark
3DMark#3	Slinghshot Test 1	3DMark
3DMark#4	Slinghshot Test 2	3DMark
GFX#1	Aztec 4K	GFXBench
GFX#2	Aztec 2K	GFXBench
GFX#3	Aztec 1080p	GFXBench
GFX#4	Manhattan 3.1 2K	GFXBench
GFX#5	Manhattan 3.1 1080p	GFXBench
GFX#6	Manhattan 3.0	GFXBench

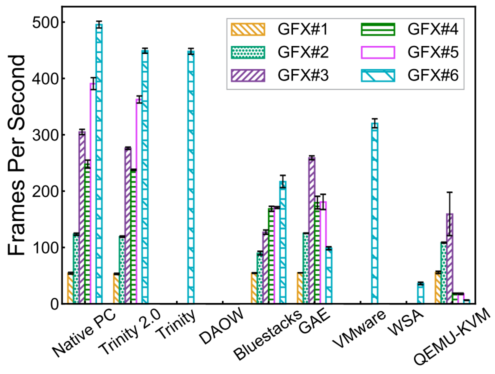
Benchmarks belonging to a same app are sorted by their workloads, i.e., the top benchmark has the heaviest workload.

Our evaluation is conducted on a high-end PC and a middle-end PC. The former has a 6-core Intel i7-8750H CPU @2.2 GHz, 16 GB RAM (DDR4 2666 MHz), and a NVIDIA GTX 1070 MAX-Q dedicated GPU. The latter has a 4-core Intel i5-9300H CPU @2.4 GHz, 8GB RAM (DDR4 2666 MHz), and an Intel UHD Graphics 630 integrated GPU. Their storage devices are both 512 GB NVME SSD. Regarding the host OS, we run most of the abovementioned emulators on Windows 11 (latest stable version) given that WSA, Bluestacks, and DAOW are Windows-specific. However, since QEMU-KVM is Linux-specific, we run it on Ubuntu 22.04 LTS which is also the latest stable version as of May. 2023.

Workloads and Methodology. We use three different workloads to drive the experiments, in order to dig out the multi-aspect performance of Trinity. First, we use representative 3D graphics benchmark applications: 3DMark [45] and GFXBench [43], both of which are widely used for evaluating mobile devices' GPU performance. Together they provide 10 specific benchmarks, as listed in Table 2. These benchmarks generate complex 3D scenes in an *off-screen* manner, that is, the rendering results are not displayed on the screen and thus is not limited by the screen's refresh rate, so the graphics system's full potential can be tested. In detail, we run each benchmark on every emulator and hardware environment for five times, and then calculate the average results



(a) 3DMark.



(b) GFXBench.

Fig. 10. Benchmark results on the high-end PC.

together with the error bars. Also, since the benchmarks come with Windows versions as well, we further run them directly on Windows to figure out the native hardware performance.

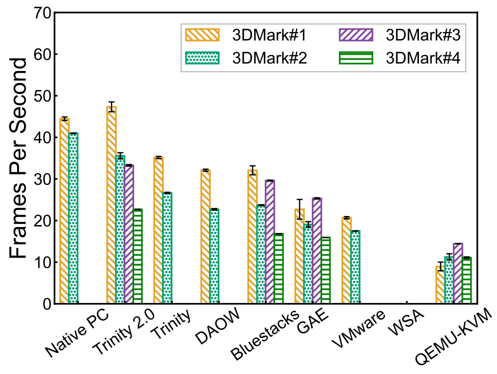
Second, to understand Trinity's performance on real apps, we run the top-100 3D (game) apps from Google Play as of 05/14/2023 [75], which are the same 100 apps discussed in Section 2.2. Concretely, for each of the apps, one of the authors manually runs a (same) full game set on every emulator, and repeats the experiment five times. During an app's running, we log the **FPS** (**Frames Per Second**) values of the app, which is a common indicator of a mobile system's running smoothness. We then use the average FPS value of the five experiments as the final FPS value of the app. Generally, we find that for all the studied apps, the standard deviations of the five experiments are all less than 4 FPS, indicating that the workloads are mostly consistent among different experiments. Since all the apps adopt the V-Sync mechanism to align their framerates with the screen's refresh rate (which is 60 Hz), their FPS values are always smaller than 60.

Third, to further evaluate Trinity's compatibility, we randomly select 10K apps from Google Play in Trinity. We use the *Monkey* UI exerciser [33] to generate random input events for each app for one minute, and monitor possible app crashes.

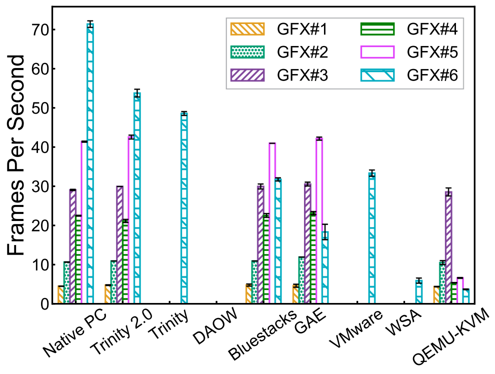
9.2 Evaluation Results

Graphics Benchmark. Figure 10 and Figure 11 illustrate the graphics benchmarks' results obtained on the high-end PC and the middle-end PC, respectively. Results of DAOW and WSA are not complete because they cannot successfully run all the benchmarks due to missing graphics APIs or abnormal API behaviors as complained by the benchmark apps. As shown, compared to the other emulators, Trinity can achieve the best efficiency on all the three benchmarks with both PCs.

Specifically, on the high-end PC that is equipped with a dedicated GPU, Trinity can outperform DAOW by an average of 40.5%, and reach 93.3% of the high-end PC's native hardware performance. In particular, for Slingshot Test 1 (3DMark#3) we can achieve 110% native performance. This is attributed to the graphics memory pool (Section 4.0.2) maintained by Trinity at the host which can fully exploit the host GPU's DMA capability. Instead, the native version of the benchmark leverages synchronous data delivery into the GPU rather than a DMA-based approach, causing suboptimal performance. Further on the middle-end PC, we observe that Trinity can outperform the other emulators by at least 12.7%, indicating that Trinity can still maintain decent efficiency even on an integrated GPU with much poorer performance.

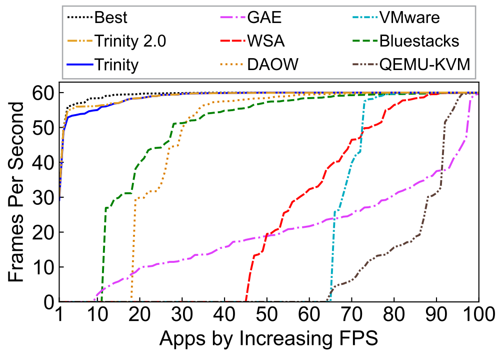


(a) 3DMark.

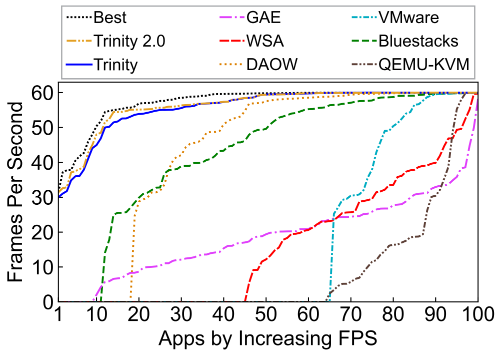


(b) GFXBench.

Fig. 11. Benchmark results on the middle-end PC.



(a) High-end PC.



(b) Middle-end PC.

Fig. 12. Average FPS of the top-100 3D apps across different emulators on the high-end and middle-end PCs. The "Best" line represents the highest FPS among the evaluated emulators of each app. If an app cannot run normally on an emulator, its corresponding FPS value is taken as zero.

Furthermore, we find that due to the enhanced reliability, Trinity 2.0 can now additionally run seven benchmarks compared to the original Trinity, which cover all the standard OpenGL ES benchmarks offered by the benchmark apps. In particular, Trinity 2.0 achieves an additional of 12.8% (26.2%) performance improvement over Trinity on the high-end (middle-end) PC, thanks to better lifecycle management and resource aggregation. The improvement benefits the middle-end PC more because the computation reduction brought by resource aggregation have more significant effects on a less powerful CPU.

Top-100 3D Apps. Figure 12 depicts the average FPS of the top-100 3D apps from Google Play on different emulator platforms, when the apps are ranked by their FPS values on the corresponding emulator. Particularly, if an app cannot be successfully executed on (i.e., is incompatible with) an emulator, its FPS value is taken as zero. Thus, the FPS values can reflect both the compatibility and efficiency of different emulators. In this regard, the original Trinity outperforms the other emulators by an average of 22.4%~538% on the evaluated PCs, while Trinity 2.0 provides an additional 0.3%——5% improvement. We next look into the compatibility and efficiency aspects of the evaluated emulators.

For compatibility, the numbers of compatible apps of Trinity (and Trinity 2.0), DAOW, Bluestacks, GAE, WSA, VMware, and QEMU-KVM are 100, 82, 89, 91, 55, 35, and 36, respectively. Delving deeper, we find that the root causes of other emulators' worse performance vary significantly. In detail, VMware, and QEMU-KVM show the worst compatibility, mostly because their guest-side graphics stacks are both built atop the open-source desktop Linux graphics library Mesa [56], whose API behaviors sometimes differ from that of a typical Android graphics library. For GAE, its incompatibility with apps in fact roots in its poor efficiency—many incompatible apps become unresponsive for a long time during a game set, thus leading to Application Not Responding (ANR) [4]. For WSA, the problem is generally the same as GAE, as we find that WSA reuses most of the GAE's host-side and guest-side system components. Differently, its lack of Google Play Service (essential for many apps' running) in the guest system introduces more compatibility issues. For Bluestacks, its stable version runs an outdated Android 7.0 guest system, and thus cannot run some recent apps. Notably, despite the selective translation of system calls (cf. Section 1) that compromises compatibility, DAOW's compatibility with the 100 game apps is only slightly worse than GAE, because it focuses on translating system calls frequently used by games [80].

For efficiency, we conduct a pairwise comparison between Trinity and each of the emulators in terms of the FPS of the apps that Trinity and the compared emulator can both successfully execute. On the high-end PC, Trinity outperforms DAOW, Bluestacks, GAE, WSA, VMware, and QEMU-KVM in terms of the compatible apps by an average of 6.1%, 9.8%, 164.8%, 34.1%, 8.6%, and 132.2%, respectively. We observe a significant visual difference between Trinity and GAE, WSA, and QEMU-KVM across all apps. We observe less visual difference between Trinity and DAOW, Bluestacks, and VMware for many apps. However, the visual difference is very noticeable especially on apps where Trinity performs more than 15 FPS better, for which there were 9, 12, and 5 apps for DAOW, Bluestacks, and VMware, respectively. Regarding the average FPS values of individual apps, we find that Trinity shows the best efficiency on 76 of the apps. For the 24 apps that Trinity shows worse efficiency, we find that the differences in the apps' average FPS values are all less than 6 FPS, with 12 of them are in fact less than 1 FPS. On these apps, we find that there is not any notable smoothness difference between Trinity and the emulators that yield the best FPS.

Similar situations can also be observed on the middle-end PC (as demonstrated in Figure 12(b)). Trinity outperforms DAOW, Bluestacks, GAE, WSA, VMware, and QEMU-KVM on the middle-end PC in terms of the compatible apps by an average of 4.9%, 16.1%, 168.7%, 84.6%, 17%, and 137.7%, respectively. Also, although there are more (42) apps where Trinity does not yield the best efficiency, the FPS differences are still mostly insignificant, with 36 of them being less than 5 FPS. For the remaining 6 apps, DAOW has the best FPS and outperforms Trinity by 6 to 9 FPS, though we could not perceive any visual difference between the two. Careful examination of the apps' runtime situations shows that they tend to heavily stress the CPU as its graphics scenes involve many physics effects such as collisions and reflections, which require the CPU to perform heavy computations such as matrix transformations. Thus, DAOW's directly interfacing with the hardware CPU without the virtualization layer allows it to perform better than Trinity (as well as the other emulators), particularly given the middle-end PC's rather weak CPU. In comparison, Trinity performs better than DAOW for all the 6 apps on the high-end PC.

Compatibility and Reliability. To examine the compatibility and reliability of both Trinity and Trinity 2.0, we conduct a series of rigorous tests with the 10K apps randomly selected from Google Play. We leverage state-of-the-art model-based UI testing techniques [30, 54] to generate input actions on the test apps to test the compatibility and reliability of Trinity and Trinity 2.0 when executing the apps. The test of an app lasts for 10 minutes and failure events (including Java/Kotlin exceptions, native crashes, and application not responding events) are continuously monitored during the process; when a failure event occurs, the test will be terminated.

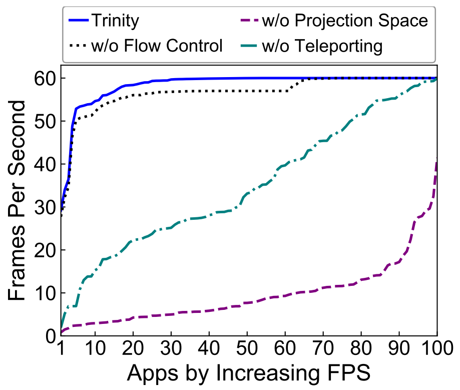


Fig. 13. Performance breakdown regarding the top-100 3D apps with framerate restriction.

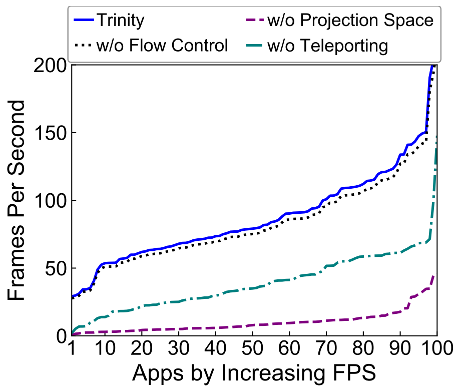


Fig. 14. Performance breakdown regarding the top-100 3D apps without framerate restriction.

For the 10K apps randomly selected from Google Play, Trinity can successfully install all of them and launch 97.2% of them. In comparison, Trinity 2.0 can launch 99.73% of them. For the apps Trinity 2.0 cannot run, we find that 0.2% require special hardware that is still under development, for example, GPS, NFC, and various sensors. Finally, the remaining 0.07% seem to actively avoid execution in an emulator by closing themselves when they notice that certain hardware configurations (e.g., the CPU specification in `/proc/cpuinfo`) are that of an emulator as complained in their runtime logs. In particular, we find that all the cross-layer reliability issues reported in production have been successfully addressed, indicating that both our analysis and enhancements are effective.

9.3 Performance Breakdown

To quantitatively understand the contributions of the proposed mechanisms to Trinity's efficiency, we respectively remove each of the three major mechanisms of Trinity (i.e., projection space, flow control and data teleporting), and measure the resulting efficiency degradations when running the top-100 3D apps on the high-end PC. In detail, removing projection space degrades Trinity to API remoting, whose guest-host control and data exchanges are still backed by our data teleporting mechanism. Removing data teleporting disables all the static timing analysis logics apart from data aggregation, which allows us to retain at least the data transferring performance of GAE since it also adopts a moderate buffer to batch void API calls. For data persistence and arrival notification, we adopt control flow blocking and VM Exit following GAE's design.

Further, to fully demonstrate the efficiency impacts of the three mechanisms, we also measure the performance breakdown when the maximum framerate restriction (which is 60 FPS) of the apps is removed. Note that we do not remove this restriction when evaluating the top-100 3D apps in Section 9.2 since this requires source code modifications to the emulators, while many of the emulators are proprietary (e.g., DAOW and Bluestacks). Figure 13 depicts the average FPS values of the top-100 3D apps in the breakdown experiments with the 60-FPS framerate restriction, while Figure 14 shows the results without the framerate restriction.

Projection Space. After the projection space is removed, the average FPS drops by 6.1× (8.6×) with (without) the framerate restriction, providing the most significant efficiency benefits. This is not surprising as our in-depth analysis of the API call characteristics (by instrumenting our system graphics library as discussed in Section 2.2 during the breakdown experiments) shows that with the projection space, 99.93% of graphics API calls do not require synchronous host-side executions. The remaining 0.07% API calls are Type-1 calls related to the context information we do not maintain in shadow contexts, including the rendered pixels and execution status of a GPU (Section 4.0.1).

Among these asynchronously-executed calls, 26% are directly resolved at the projection space (with our maintained context and resource information), fundamentally avoiding their needs for any host-side executions. Such calls are mostly related to context manipulation and context/resource information querying. For the remainder (74%), they involve APIs for resource allocations and populations, as well as drawing calls. We also measure the memory consumption of the added projection space when running the top-100 3D apps by monitoring the maximum memory consumed by our provided system graphics library at the guest side. We find that the projection space only takes an average of 466 KB (at most 1021 KB) memory for an app. The memory consumption is small because the shadow contexts and resource handles are mostly small integers, and our careful resource management has prevented redundant memory usages.

Flow Control. On the other hand, flow control contributes 2.7% (5%) FPS improvement on average with (without) the framerate restriction. This is because flow control mainly serves to mitigate the control flow oscillation problem (cf. Section 5), thus contributing less to the running smoothness *as measured by FPS*. To quantify the actual effects of flow control, we further measure the occurrences of control flow oscillations during the apps' running. As a result, without flow control, control flow oscillation occurs 20× more frequently on average. When that happens, as discussed in Section 5, the apps' animations will look extremely unsmooth *from users' perspective* since many essential frames of the animations are skipped (i.e., not rendered) by the apps as dictated by the delta timing principle, while the total number of frames rendered per second (i.e., FPS) remains mostly unchanged.

Data Teleporting. Finally, when data teleporting is disabled, the fixed data delivery strategy cannot adapt to system and data dynamics, leading to 1.7× (2.2×) FPS degradation with (without) the framerate restriction. To demystify the efficiency gains brought by data teleporting, we further examine its throughput under diverse system and data dynamics on the high-end PC. Specifically, we develop a benchmark app that synthesizes data chunks ranging from 4 KB (a continuous memory page space) to 128 MB, and doubles the size for each successive experiment. In each experiment, the app writes the data chunk to our kernel character device file (cf. Section 8) to transfer it to the host 1,000 times with one, two, three, or four threads; here the number of threads varies from one to four (the number of the emulator's CPU cores) to mimic different system dynamics. By measuring the time consumed for data transfer, we can calculate the final throughput result.

In comparison, we conduct the same experiments on GAE's guest-host I/O pipe `goldfish-pipe`, which is GAE's core infrastructure for sending API call data from the guest to the host and realizing API remoting. To this end, we customize GAE to include a dedicated graphics API for throughput measurement, which our benchmark app can call to transfer guest data to the host as described above. This API is made to be a void API so that GAE's buffer for batching void APIs can take effect. Consequently, as shown in Figure 15 and Figure 16, data teleporting's throughput clearly exceeds that of `goldfish-pipe` under all the data and thread settings. On average, data teleporting's throughput is 5.3 times larger than that of `goldfish-pipe`.

Furthermore, we wish to know the effectiveness of static timing analysis. For this purpose, we measure the performance of the data teleporting mechanism using the above experiments when we adopt every possible strategy. Then, we compare the highest throughput produced by the above strategy exhaustion with that produced by the static timing analysis. As shown in Figure 17 and Figure 18, the throughput values produced by strategy exhaustion and static timing analysis are

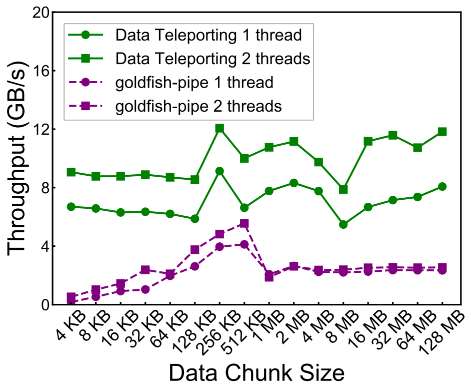


Fig. 15. Throughput of data teleporting and goldfish-pipe, with one and two threads.

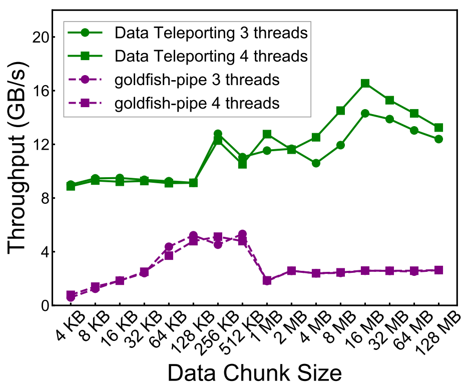


Fig. 16. Throughput of data teleporting and goldfish-pipe, with three and four threads.

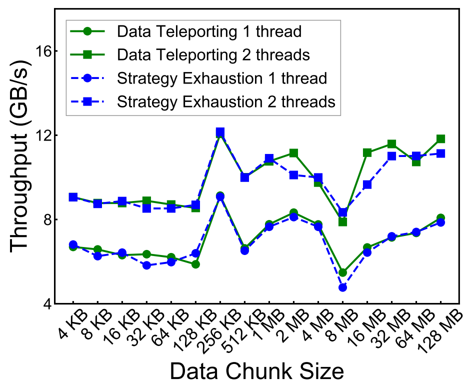


Fig. 17. Throughput of data teleporting using strategy exhaustion and static timing analysis, with one and two threads.

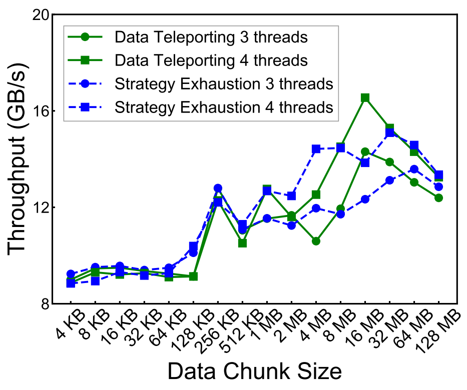


Fig. 18. Throughput of data teleporting using strategy exhaustion and static timing analysis, with three and four threads.

very close (4% average deviation). More in detail, static timing analysis can make the most suitable strategy choice in 95.4% of the data delivery tasks.

10 RELATED WORK

Commercial Mobile Emulators. A plethora of commercial mobile emulators have similar architectures to the ones we evaluate in Section 9. For instance, Anbox [3], which directly runs Android's Framework layer on a Linux PC, leverages the container technique to achieve lightweight guest-host isolation, and reuses GAE's graphics stack—all the guest-side graphics operations are sent to a host-side daemon for execution, thus requiring synchronous inter-process communications. Accordingly, its efficiency is similar to that of GAE.

LDPlayer [46], MEmu [55], NoxPlayer [61], and Genymotion [28] all adopt the AOVB (Android-x86 on VirtualBox) architecture (as in Bluestacks). To realize graphics rendering, they also reuse some of the graphics libraries of GAE, for example, libGLESv2_enc at the guest that encodes OpenGL ES API calls into a data packet, and ANGLE [31] at the host that translates guest-side OpenGL ES calls to desktop OpenGL or Direct3D calls. Prior measurements [14, 80] show that the performance of such AOVB-based emulators is close to that of Bluestacks, probably due to their similar architectures.

GPU Virtualization. In PC/server virtualization, GPU multiplexing is typically achieved through hardware-assisted GPU passthrough [1, 2] or mediated passthrough [38, 41, 62], which allow a VM to directly access the host GPU by remapping its DMA channels and interrupts to the guest. Differently, GPU passthrough monopolizes the host GPU, while the mediated approach allows sharing the GPU among multiple VMs through GPU context isolation.

However, the substantial differences between the graphics stacks of desktop OSes and mobile OSes significantly hinder their adoption by mobile emulators, as host GPUs' drivers are missing in mobile systems and developing them for mobile environments is extremely complicated (since mainstream desktop GPUs' specifications are often proprietary). Hence, we take a completely different approach of graphics projection to address the problem of multiplexing the host GPU, which is agnostic to the underlying hardware specifications and thus should also be beneficial to PC/server GPU virtualization.

Cross-OS and Cross-Device Graphics Stacks. Trinity focuses on Android emulation on a PC, while several researches have explored running iOS apps on Android graphics stacks based on their similarities in OpenGL ES libraries [8, 9]. This suggests that Trinity's graphics projection mechanism might also be applicable to the emulation of iOS apps on a PC. Also, various approaches remote graphics processing from one device to another over a network [10–12, 34, 68, 70]. For them, data exchanges over network often constitute a major bottleneck, which is similar to the bottleneck of frequent cross-boundary control/data exchanges in the virtualization setting. Thus, our idea of decoupling guest/host control and data flows via graphics projection should also be useful to relevant studies and applications, for example, cloud/edge gaming.

Cross-Layer and Cross-System Failures. Several recent studies [37, 52, 73] have paid special attention to cross-layer and cross-system failures in cloud and networking systems, due to their increasingly decoupled components and microservices. Tang et al. [73] conduct the first comprehensive study of **cross-system interaction** (**CSI**) failures in production cloud systems, which are failures induced by faulty interactions among decoupled systems. They identify the major root cause of CSI failures as the discrepancies in the systems' control and data planes, and provide insights for testing and preventing the failures. Liu et al. [52] also uncover that one of the major root causes of cloud system failures is the inconsistent data format among interaction layers/systems. These studies show that cross-layer and cross-system failures have become severe and prevalent in practice. In this work, we focus on understanding and diagnosing a similar problem in our virtualized graphics system, offering essential insights and experiences for other virtualization systems and beyond.

11 CONCLUSION

In this article, we present the design, implementation, performance, and preliminary deployment of the Trinity mobile emulator. It substantially boosts the efficiency of mobile emulation while retaining high compatibility and security through graphics projection, a novel approach that minimizes the coupling between the guest-side and host-side graphics processing. This unique design, together with strategic flow control and data teleporting, make Trinity a first-of-its-kind emulator that can smoothly run heavy 3D mobile games (achieving near-native hardware performance) and meanwhile retain comprehensive app support and solid guest-host isolation.

As part of a major commercial Android IDE, Trinity is expected to be used by millions of Android developers in the near future, contributing vibrantly to the ecosystem. To meet the strict production requirement of reliability, we develop a root cause analysis pipeline for diagnosing cross-layer reliability issues that plague Trinity. Our pipeline helps identify major reliability issues and facilitate problem fixing, which leads to strengthened runtime reliability and enhanced performance. We believe that many lessons and experiences gained from this work could also be applied to (graphics-heavy) PC emulation and cloud/edge systems, as to be explored in our future work.

REFERENCES

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel virtualization technology for directed I/O. *Intel Technology Journal* (2006).

[2] AMD. 2009. I/O virtualization technology specification revision 1.26. *AMD White Paper* 1 (2009), 2–11.

[3] Anbox.com. 2021. Anbox: Container-Based Android Emulator. Retrieved from https://anbox.io/

[4] Android.org. 2021. Application Not Responding of Android. Retrieved from https://developer.android.com/topic/performance/vitals/anr

[5] Android.org. 2021. GraphicBuffer: Android's Native Window Buffer Implementation. Retrieved from https://android.googlesource.com/platform/frameworks/native/+/jb-mr0-release/libs/ui/GraphicBuffer.cpp

[6] Android.org. 2021. View: Basic Building Blocks for Android User Interface. Retrieved from https://developer.android.com/reference/android/view/View

[7] Android.org. 2023. Android System Tracing Tool. Retrieved from https://developer.android.com/topic/performance/tracing

[8] Jeremy Andrus, Naser AlDuaij, and Jason Nieh. 2017. Binary compatible graphics support in android for running iOS apps. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 55–67.

[9] Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij, Christoffer Dall, Nicolas Viennot, and Jason Nieh. 2014. Cider: Native execution of IOS apps on android. In *Proceedings of the ACM ASPLOS*. 367–382.

[10] Apple.com. 2021. AirPlay: Share Mutimedia Contents across Devices. Retrieved from https://www.apple.com/airplay/

[11] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. 2005. THINC: A virtual display architecture for thin-client computing. In *Proceedings of the ACM SOSP*. 277–290.

[12] Ricardo A. Baratto, Shaya Potter, Gong Su, and Jason Nieh. 2004. MobiDesk: Mobile virtual desktop computing. In *Proceedings of the ACM MobiCom*. 1–15.

[13] David Blaauw, Kaviraj Chopra, Ashish Srivastava, and Lou Scheffer. 2008. Statistical timing analysis: From basic principles to state of the art. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 4 (2008), 589–607.

[14] Bluestacks.com. 2021. Benchmark Performance Comparisions Among Bluestacks, LDPlayer, Memu, and Nox. Retrieved from https://www.bluestacks.com/bluestacks-vs-ldplayer-vs-memu-vs-nox.html

[15] Bluestacks.com. 2021. Bluestacks: Modern Android Gaming Emulator. Retrieved fromhttps://www.bluestacks.com/

[16] T. Capin, K. Pulli, and T. Akenine-Moller. 2008. The state of the art in mobile graphics research. *IEEE Computer Graphics and Applications* 28, 4 (2008), 74–84.

[17] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic replay of java multithreaded applications. In *Proceedings of the ACM SPDT*. 48–59.

[18] Shane Cook. 2012. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes.

[19] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE S&P*. 297–312.

[20] Micah Dowty and Jeremy Sugerman. 2009. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 73–82.

[21] DPDK.org. 2023. rte_ring: Lock-Free Ring Buffer in DPDK. Retrieved from https://doc.dpdk.org/api/rte__ring_8h.html

[22] José Duato, Antonio J. Pena, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the IEEE HPC*. 224–231.

[23] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224.

[24] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. 2008. Execution replay of multiprocessor virtual machines. In *Proceedings of the ACM VEE*. 121–130.

[25] Anne Edmundson, Roya Ensafi, Nick Feamster, and Jennifer Rexford. 2016. A first look into transnational routing detours. In *Proceedings of the ACM SIGCOMM*. 567–568.

[26] Facebook.com. 2023. Zstandard: Fast Real-time Compression Algorithm. Retrieved from http://facebook.github.io/zstd/

[27] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. 2022. Trinity: High-performance mobile emulation through graphics projection. In *Proceedings of the USENIX OSDI*. 285–301.
[28] Genymotion.com. 2021. Genymotion: Android as a Service. Retrieved from https://www.genymotion.com/
[29] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu. 2020. Experiences of landing machine learning onto market-scale mobile malware detection. In *Proceedings of the ACM EuroSys*. 1–14.
[30] Liangyi Gong, Hao Lin, Zhenhua Li, Feng Qian, Yang Li, Xiaobo Ma, and Yunhao Liu. 2020. Systematically landing machine learning onto market-scale mobile malware detection. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1615–1628.
[31] Google.com. 2021. Almost Native Graphics Layer Engine. Retrieved from https://github.com/google/angle
[32] Google.com. 2021. Android Emulator: Simulates Android Devices on Your Computer. Retrieved from https://developer.android.com/studio/run/emulator
[33] Google.com. 2021. Monkey: Automatic UI/Application Exerciser. Retrieved from https://developer.android.com/studio/test/monkey
[34] Google.com. 2021. Stream Content with Chromecast. Retrieved from https://store.google.com/us/product/chromecast?hl=en-US
[35] Google.com. 2021. SwiftShader: A CPU-Based Implementation of Graphics APIs. Retrieved from https://github.com/google/swiftshader
[36] Google.com. 2023. Google Firebase Performance Monitoring Tool. Retrieved from https://firebase.google.com/docs/perf-mon
[37] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What bugs live in the cloud?. In *Proceedings of the ACM SoCC*. 1–14.
[38] Alex Herrera. 2014. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. *NVIDIA Corp* (2014), 1–18.
[39] Huawei.com. 2021. Huawei's DevEco Studio. Retrieved from https://developer.harmonyos.com/en/develop/deveco-studio/
[40] Intel.com. 2021. Houdini: Translate The ARM Binary Code Into the x86 Instruction Set. Retrieved from https://www.intel.com/content/www/us/en/products/docs/workstations/resources/accelerate-game-development-houdini-optane-memory.html
[41] Intel.com. 2021. Intel GVT-g: Full GPU Virtualization with Mediated Pass-through. Retrieved from https://github.com/intel/gvt-linux/wiki/GVTg_Setup_Guide
[42] Tom Kelly. 2003. Scalable TCP: Improving performance in highspeed wide area networks. In *Proceedings of the ACM SIGCOMM*. 83–91.
[43] Kishonti Ltd. 2021. GFXBench: A Unified Graphics Benchmark Based on DXBenchmark. Retrieved from https://gfxbench.com/
[44] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: The linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Vol. 1. 225–230.
[45] Underwriters Laboratories. 2021. 3DMark: Popular Benchmarks for Gamers, Overclockers, and System Builders. Retrieved from https://www.3dmark.com/
[46] LDPlayer.com. 2021. LDPlayer: Free Android Emulator for PC. Retrieved from https://www.ldplayer.net/
[47] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. Chimera: Hybrid program analysis for determinism. In *Proceedings of the ACM PLDI*. 463–474.
[48] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the ACM ASPLOS*. 77–90.
[49] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the ACM MobiSys*. 151–165.
[50] Mingliang Li, Hao Lin, Cai Liu, Zhenhua Li, Feng Qian, Yunhao Liu, Nian Sun, and Tianyin Xu. 2020. Experience: Aging or glitching? why does android stop responding and what can we do about it?. In *Proceedings of the ACM MobiCom*. 1–11.
[51] Linux.org. 2023. kfifo: Lock-Free Ring Buffer in Linux Kernel. Retrieved from https://archive.kernel.org/oldlinux/htmldocs/kernel-api/kfifo.html
[52] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on HotOS*. 155–162.

[53] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. 2022. Hubble: Performance debugging with in-production, just-in-time method tracing on android. In *Proceedings of the USENIX OSDI*. 787–803.

[54] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable automated model-based GUI testing for android enhanced by reinforcement learning. In *Proceedings of the IEEE/ACM ASE*. 5.

[55] MEmu.com. 2021. MEmu: The Most Powerful Android Emulator. Retrieved from https://www.memuplay.com/

[56] Mesa.org. 2021. The Mesa 3D Graphics Library. Retrieved from https://www.mesa3d.org/

[57] Microsoft.com. 2021. Introduction to Hyper-V on Windows. Retrieved from https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/

[58] Microsoft.com. 2021. Windows Subsystem for Android. Retrieved from https://docs.microsoft.com/en-us/windows/android/wsa/

[59] Ravi Netravali and James Mickens. 2019. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM SoCC*. 428–440.

[60] Matthias Neugschwandtner, Christian Platzer, Paolo Milani Comparetti, and Ulrich Bayer. 2010. dAnubis–dynamic device driver analysis based on virtual machine introspection. In *Proceedings of the Springer DIMVA*. Springer, 41–60.

[61] NoxPlayer.com. 2021. NoxPlayer: The Perfect Android Emulator to Play Mobile Games on PC. Retrieved from https://www.bignox.com/

[62] NVIDIA.com. 2021. vGPU: Security Benefits of Virtualization as well as the Performance of NVIDIA GPUs. Retrieved from https://www.nvidia.com/en-us/data-center/virtual-solutions/

[63] Jon Oberheide and Charlie Miller. 2012. Dissecting the android bouncer. *SummerCon2012, New York* 95 (2012), 110.

[64] Oracle.com. 2021. VirtualBox: A Powerful x86 and AMD64/Intel64 Virtualization Product. Retrieved from https://www.virtualbox.org/

[65] Bryan D Payne, DP de A Martim, and Wenke Lee. 2007. Secure and flexible monitoring of virtual machines. In *Proceedings of the IEEE ACSAC*. 385–397.

[66] QEMU.org. 2021. QEMU: A Generic and Open Source Machine Emulator and Virtualizer. Retrieved from https://www.qemu.org/

[67] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the USENIX OSDI*. 107–120.

[68] RealVNC.com. 2021. VNC; Remote Desktop Access. Retrieved from https://www.realvnc.com/en/

[69] RenderDoc.org. 2023. RenderDoc: A Standalone Graphics Debugging Tool for Vulkan, OpenGL, and OpenGL ES. Retrieved from https://renderdoc.org/

[70] Shu Shi and Cheng-Hsin Hsu. 2015. A survey of interactive remote rendering systems. *Computing Surveys* 47, 4 (2015), 1–29.

[71] Skia.org. 2021. Skia: 2D Graphics Rendering Library. Retrieved from https://skia.org/

[72] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why not virtualizing gpus at the hypervisor?. In *Proceedings of the USENIX ATC*. 109–120.

[73] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. 2023. Fail through the Cracks: Cross-system interaction failures in modern cloud systems. In *Proceedings of the ACM EuroSys*. 433–451.

[74] Trinity. 2023. Bug Report & Fix: Problematic API Definations of OpenGL ES. Retrieved from https://github.com/KhronosGroup/OpenGL-Refpages/pull/118

[75] Trinity.github. 2021. List of Top-100 3D Apps. Retrieved from https://github.com/TrinityEmulator/EvaluationScript/#4-top-100-3d-apps

[76] VMware.com. 2021. VMware Workstation Pros: Run Windows, Linux and BSD Virtual Machines on a Windows or Linux Desktop. Retrieved from https://www.vmware.com/products/workstation-pro.html

[77] Wm A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.

[78] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the USENIX Security*. 569–584.

[79] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and detecting overlay-based android malware at market scales. In *Proceedings of the ACM MobiSys*. 168–179.

[80] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile gaming on personal computers with direct android emulation. In *Proceedings of the ACM MobiCom*. 1–15.

Received 13 June 2023; revised 13 June 2023; accepted 9 January 2024