Trinity: High-Performance Mobile Emulation through Graphics Projection

Di Gao[†*], Hao Lin[†*], Zhenhua Li[†], Chengen Huang[†], Yunhao Liu[†]
Feng Qian[§], Liangyi Gong[‡], Tianyin Xu[¶]
[†]*Tsinghua University* [§]*University of Minnesota* [‡]*CNIC, CAS* [¶]*UIUC*

Abstract

Mobile emulation, which creates full-fledged software mobile devices on a physical PC/server, is pivotal to the mobile ecosystem, especially for PC-based mobile gaming, app debugging, and malware detection. Unfortunately, existing mobile emulators perform poorly on graphics-intensive apps in terms of either efficiency or compatibility or both. To address this, we introduce *graphics projection*, a novel graphics virtualization mechanism that adds a small-size *projection space* inside the guest memory of a virtual mobile device. The projection space processes graphics operations involving control contexts and resource handles without host interactions. Novel flow control and data teleporting mechanisms are devised to match the decoupled graphics processing rates of the virtual device and the host GPU to maximize performance. The resulting new Android emulator, dubbed Trinity, exhibits an average of 93.3% native hardware performance and 97.2% app support, in some cases outperforming other emulators by more than an order of magnitude. It has been adopted by Huawei DevEco Studio, a major Android IDE with millions of developers.

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 [14] and DAOW [55]) are used to detect malware in app markets [21, 44, 54], to enable mobile gaming on PCs [14, 55], and to empower the emerging notion of cloud gaming [36].

1.1 Motivation

To create full-fledged software mobile devices on a physical PC/server, mobile emulators usually adopt the classic virtualization framework [33, 40, 45, 46] where a mobile OS runs in a virtual machine (VM), referred to as the guest, hosted on a PC/server, referred to as 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 [37] and thus expecting mobile emulators to have powerful graphics processing capabilities (as provided by real mobile phones) [55]. This *capability gap* is further aggravated by the substantial architectural differences between the graphics stacks of desktop and mobile OSes [15].

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 [26] 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 [18, 50], 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 [19] 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* [17] 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 [17], 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.

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

Another approach is to break guest-host isolation by removing the virtualization layer so apps can directly use the GPU, as embodied in DAOW [55]. 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.

1.2 Contribution

We present Trinity, a novel mobile emulator that simultaneously achieves high efficiency and compatibility. Our guiding principle is to decouple the guest-host control and data 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 graphics 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, *etc.*; 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, *i.e.,* 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 in Trinity to orchestrate the execution speeds of 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 are common in graphics operations). To this end, we find that the dynamic situations in fact 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* [12]

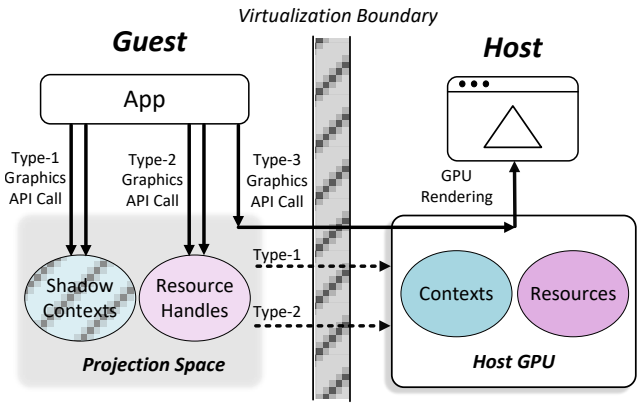


Figure 1: Basic workflow of Trinity.

to estimate which strategy is best suited to a data flow. With these efforts, we achieve high emulation efficiency for Trinity.

Similar to GAE, Trinity is also implemented atop QEMU (for general device extensibility) and hosts the Android OS, with 118K lines of C/C++ code. We evaluate its performance 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, VMware Workstation, Bluestacks, and DAOW. The evaluation shows that Trinity can achieve 80%∼110% (averaging at 93.3%) native hardware performance, outperforming the other emulators by 1.4× to 20×. For compatibility, Trinity can run the top-100 3D apps and 97.2% 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 (or security).

Software/Code/Data Availability. Trinity has recently been adopted by Huawei DevEco Studio [28], a major Android IDE (integrated development environment) with millions of developers. Currently, it is going through the beta test run for minor functional adjustments and bug fixes. The binary, code, and measurement data involved in this work are released at `https://TrinityEmulator.github.io/`.

2 Understanding Mobile Graphics APIs

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

2.1 Background

Figure 2 shows a basic OpenGL ES program for drawing a triangle. The program creates a graphics buffer in a GPU's graphics memory using a Type-2 API—`glGenBuffers`, populates the buffer with the coordinate data of the triangle's

```
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 bound 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();
    ...
    // Type-3: draw the triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);
    ...
}
```

2. All subsequent operations do not need to specify the handle again

(b) Draw the triangle.

Figure 2: OpenGL ES code snippet for drawing a triangle.

vertices through a Type-1 API—`glBindBuffer` and a Type-2 API—`glMapBufferRange`, and then 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*, *i.e.,* 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.

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, *e.g.,* 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, *e.g.,* whether to perform validation of pixel values after a frame is rendered. Resource attributes correspond to resources' internal information, *e.g.,* 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, *i.e.,* the buffer underlying `vertex_buffer_handle`. This approach is easy to implement but involves synchronous, time-consuming memory copies. In contrast, Figure 2 shows the latent mapping approach, where `glBufferData` is called but no data are passed to it; `glMapBufferRange` instead maps the graphics buffer to a main memory address, *i.e.,* `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. The data are latently copied to the graphics buffer by the GPU's hardware *copy engine* (a DMA device) usually when `glUnmapBuffer` is called to release the address mapping, thus being 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, *e.g.,* `glDrawArrays` as shown in Figure 2. 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 [16] 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.

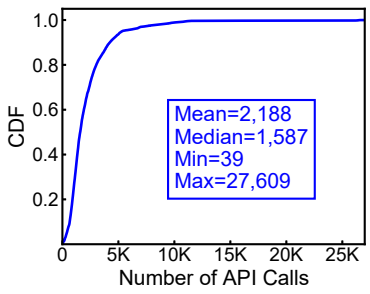


Figure 3: Number of API calls issued for rendering a single frame.

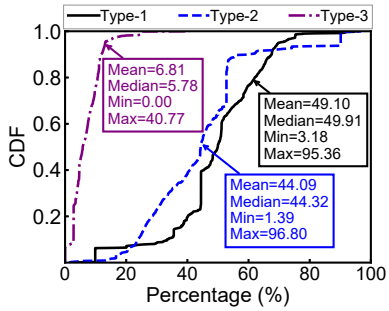


Figure 4: Percentages of specific types of API calls for the top-100 3D apps.

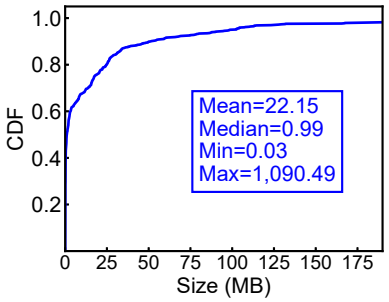


Figure 5: Graphics data amount generated per second by top-100 3D apps.

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 [53], *i.e.,* 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 11/20/2021 [51] 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, 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, *i.e.,* 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, *i.e.,* 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.

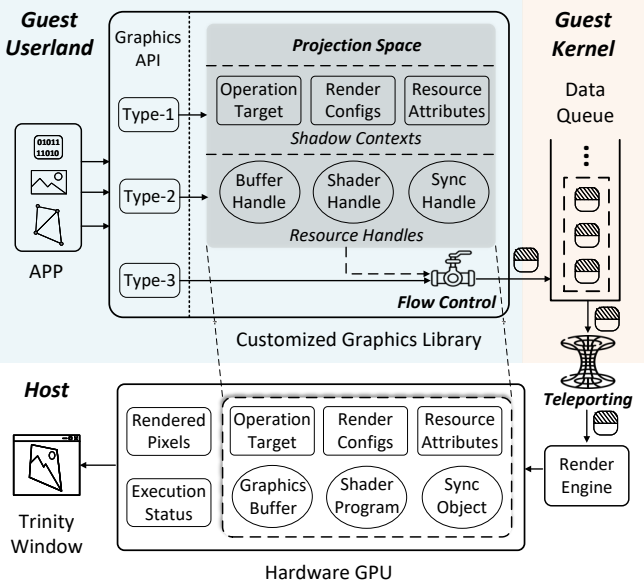


Figure 6: Architectural overview of Trinity.

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.* §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 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.* §6.1), 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.* §8.3). 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 (§5). Furthermore, the projection-host interactions are accomplished via a data teleporting method (§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 (§4.1) and resource handles (§4.2), *i.e.,* the key data structures that format the projection space.

4.1 Shadow Context

In §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, *i.e.,* 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 §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 §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.

4.2 Resource Handle

As introduced in §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 §2.1, there are two approaches to populating a graphics buffer with data, *i.e.,* 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 §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 (*cf.* §2.1). After this, 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 §2.1. As a result, the guest process may generate abnormal animations following the delta timing principle, *e.g.,* 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 [31] 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 (§6.1) that constitute a major obstacle to the goal, and then describe the workflow of our data teleporting method (§6.2), which leverages static timing analysis to accommodate the dynamic situations.

6.1 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

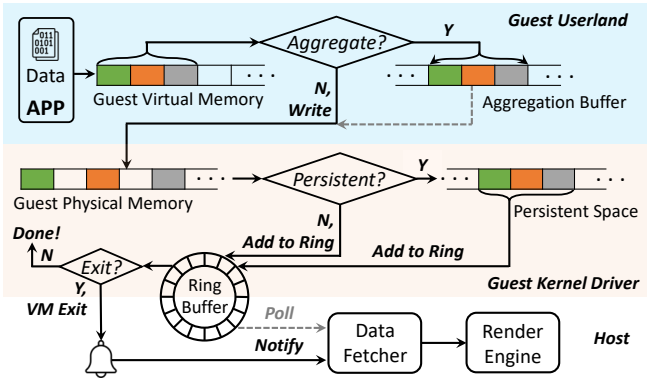


Figure 7: Workflow of data teleporting.

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, *i.e., 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.

6.2 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* [12] 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.* §7), 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 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 1024) 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, *i.e.,* $T_{persistent} = \frac{S_{data}}{V_{guest}} + T_{ring}$. 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 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, 49] 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, *e.g.,* `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 [29] into the guest system for dynamic binary translation.

8 Evaluation

We evaluate Trinity with regard to our goal of simultaneously maintaining high efficiency and compatibility. First, we describe our experiment setup in §8.1. Next, we present the evaluation results in §8.2, 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 with 10K apps randomly selected from Google Play. Finally, we present the performance breakdown in §8.3 by removing each of the three major system mechanisms—projection space, flow control and data teleporting.

8.1 Experiment Setup

To understand the performance of Trinity in a comprehensive manner, we compare it with six mainstream emulators, including GAE, QEMU-KVM, VMware Workstation, Bluestacks, and DAOW, as well as Windows Subsystem for Android (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 Dec. 2021.

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.

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 20.04 LTS which is also the latest stable version as of Dec. 2021.

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 [34] and GFXBench [32], both of which are widely used for evaluating mobile devices' GPU performance. Together they provide three specific benchmarks, which are referred to as Slingshot Unlimited Test 1 (3DMark), Slingshot Unlimited Test 2 (3DMark) and Manhattan Offscreen 1080p (GFXBench).

Table 1: Comparison of the evaluated emulators.

Mobile Emulator	System Architecture	Graphics Stack
GAE [23]	x86 Android on customized QEMU	API remoting
WSA [41]	x86 Android on Windows Hyper-V	API remoting
QEMU-KVM [46]	Android-x86 on QEMU	Device emulation
VMware Workstation [52]	Android-x86 on VMware Workstation	Device emulation
Bluestacks [14]	Android-x86 on VirtualBox	Proprietary
DAOW [55]	Direct Android emulation on Windows	API translation with ANGLE [22]

These benchmarks generate complex 3D scenes in an *off-screen* manner, *i.e.,* 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 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 11/20/2021 [51], which are the same 100 apps discussed in §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 [24] to generate random input events for each app for one minute, and monitor possible app crashes.

8.2 Evaluation Results

Graphics Benchmark. Figure 8 and Figure 9 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.

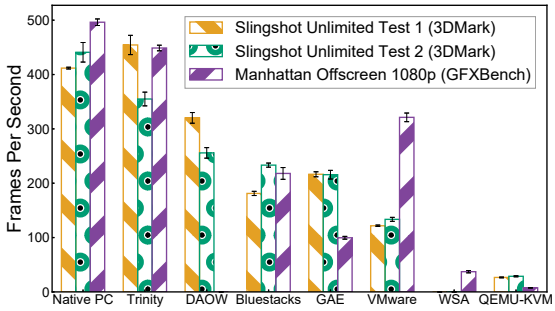


Figure 8: Benchmark results on the high-end PC.

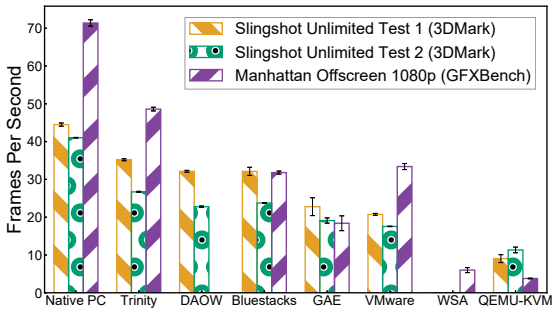
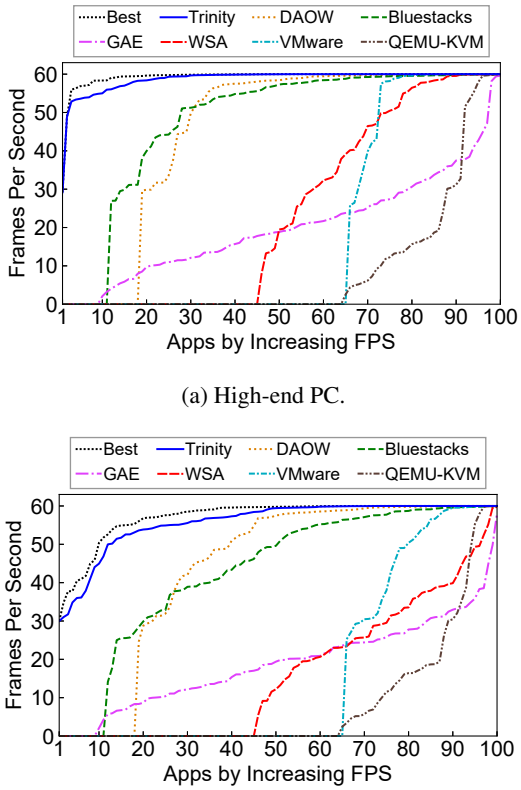


Figure 9: Benchmark results on the middle-end PC.



(a) High-end PC.

(b) Middle-end PC.

Figure 10: 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.

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 Unlimited Test 1 we can achieve 110% native performance. This is attributed to the graphics memory pool (§4.2) 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.

Top-100 3D Apps. Figure 10 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, Trinity outperforms the other emulators by an average of 22.4%∼538% on the evaluated PCs. We next look into the compatibility and efficiency aspects of the evaluated emulators, respectively.

For compatibility, the numbers of compatible apps of Trinity, 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 [39], 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.* §1.1) 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 [55].

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 10b). 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 with Random 10K Apps. For the apps randomly selected from Google Play, we can successfully install all of them and run 97.2% of them without incurring app crashes. For the apps we cannot run, we find that some (2.3%) of them have also exhibited crashes on real devices; In addition, 0.43% require special hardware that Trinity currently has not implemented, *e.g.,* GPS, NFC and various sensors, which is not hard to fix given the general device extensibility of QEMU that Trinity is built on. Finally, the remaining 0.07% seem to actively avoid being run in an emulator by closing themselves when they notice that certain hardware configurations (*e.g.,* the CPU specification listed in `/proc/cpuinfo`) are that of an emulator as complained in their runtime logs.

8.3 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 §8.2 since this requires source code modifications to the emulators, while many of the emulators are proprietary (*e.g.,* DAOW and Bluestacks). Figure 11 depicts the average FPS values of the top-100 3D apps in the breakdown experiments with the 60-FPS framerate restriction, while Figure 12 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 §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 as discussed in §4.1.

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

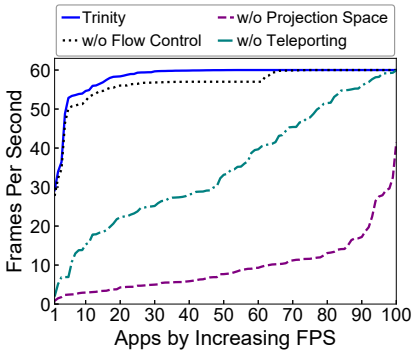


Figure 11: Performance breakdown with regard to the top-100 3D apps with framerate restriction.

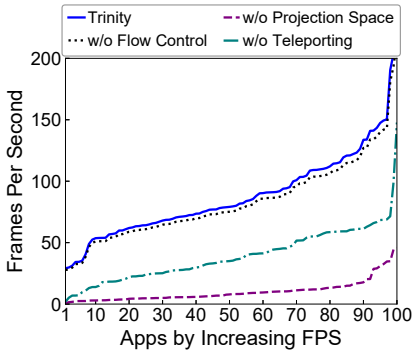


Figure 12: Performance breakdown with regard to the top-100 3D apps without framerate restriction.

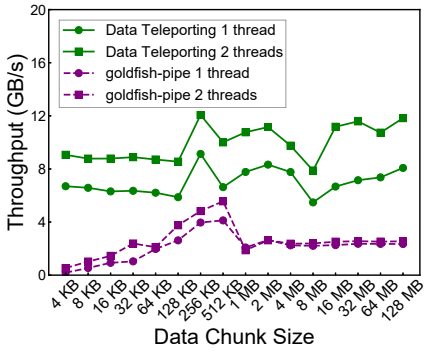


Figure 13: Throughput of data teleporting and `goldfish-pipe`, with one and two threads.

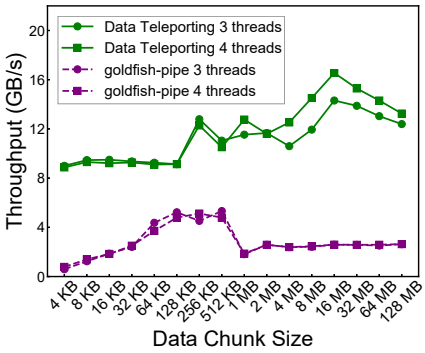


Figure 14: Throughput of data teleporting and `goldfish-pipe`, with three and four threads.

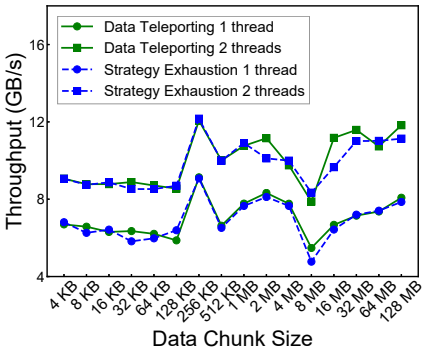


Figure 15: Throughput of data teleporting using strategy exhaustion and static timing analysis, with one and two threads.

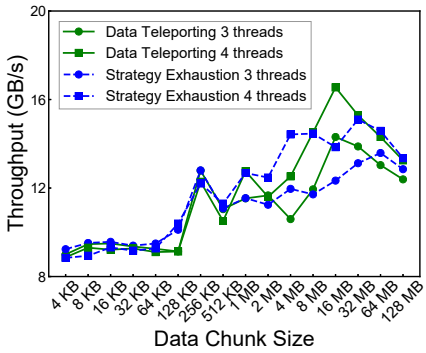


Figure 16: Throughput of data teleporting using strategy exhaustion and static timing analysis, with three and four threads.

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.* §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 §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 well 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.* §7) 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 called `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 13 and Figure 14, 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 15 and Figure 16, the throughput values produced by strategy exhaustion and static timing analysis are 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.

9 Related Work

Commercial Mobile Emulators. A plethora of commercial mobile emulators have similar architectures to the ones we evaluate in §8. 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 [35], MEmu [38], NoxPlayer [42] and Genymotion [20] 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, *e.g.,* `libGLESv2_enc` at the guest that encodes OpenGL ES API calls into a data packet, and ANGLE [22] at the host that translates guest-side OpenGL ES calls to desktop OpenGL or Direct3D calls. Prior measurements [13, 55] 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 [27, 30, 43], which allow a virtual machine (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 [7, 8]. 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 [9–11, 25, 47, 48]. 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, *e.g.,* cloud/edge gaming.

10 Conclusion

In this paper 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. We believe that many lessons and experiences gained from this work could also be applied to (graphics-heavy) PC emulation and cloud/edge gaming, as to be explored in our future work.

Acknowledgements

We would like to express our deepest appreciation to our shepherd, Jason Nieh, who was very responsive during our interactions with him and provided us with valuable suggestions, which have significantly improved our paper. We also thank the anonymous reviewers for their constructive suggestions. We thank Wei Liu and Xinlei Yang for their help in data collection and analysis. This work is supported in part by the National Key R&D Program of China under grant 2021YFB2900100, as well as the National Natural Science Foundation of China (NSFC) under grant 61902211.

A Artifact Appendix

Abstract

Trinity's artifact is publicly available at GitHub. To facilitate developing and using Trinity, we provide step-by-step instructions in the form of both documentations and videos. Please refer to our README file at https://github.com/TrinityEmulator/TrinityEmulator for details.

Scope

The artifact can be used to reproduce all the major results, including those of the graphics benchmarks and 3D apps.

Contents

Trinity's artifact includes code of the host emulator, binary of the guest Android system, and our evaluation scripts/data.

Hosting

We host the code/binary and data in two repositories (both in the `main` branch). We also provide a DOI for the artifact.

- **Trinity Code and Binary.**

  Link: https://github.com/TrinityEmulator/TrinityEmulator.

- **Evaluation Data and Figure Plotting Script.**

  Link: https://github.com/TrinityEmulator/EvaluationScript.

- **DOI for the Artifact.**

  DOI: 10.5281/zenodo.6586575

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 2006.

[2] AMD. I/O Virtualization Technology Specification Revision 1.26. *AMD White Paper*, 1:2–11, 2009.

[3] Anbox.com. Anbox: Container-based Android Emulator, 2021. https://anbox.io/.

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

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

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

[7] J. Andrus, N. AlDuaij, and J. Nieh. Binary Compatible Graphics Support in Android for Running iOS Apps. In *Proc. of ACM/IFIP/USENIX Middleware*, pages 55–67, 2017.

[8] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh. Cider: Native Execution of IOS Apps on Android. In *Proc. of ACM ASPLOS*, pages 367–382, 2014.

[9] Apple.com. AirPlay: Share Mutimedia Contents across Devices, 2021. https://www.apple.com/airplay/.

[10] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proc. of ACM SOSP*, pages 277–290, 2005.

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

[12] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer. Statistical Timing Analysis: From Basic Principles to State of The Art. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):589–607, 2008.

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

[14] Bluestacks.com. Bluestacks: Modern Android Gaming Emulator, 2021. https://www.bluestacks.com/.

[15] T. Capin, K. Pulli, and T. Akenine-Moller. The State of the Art in Mobile Graphics Research. *IEEE Computer Graphics and Applications*, 28(4):74–84, 2008.

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

[17] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[18] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *Proc. of IEEE HPC*, pages 224–231, 2010.

[19] A. Edmundson, R. Ensafi, N. Feamster, and J. Rexford. A First Look into Transnational Routing Detours. In *Proc. of ACM SIGCOMM*, pages 567–568, 2016.

[20] Genymotion.com. Genymotion: Android as a Service, 2021. https://www.genymotion.com/.

[21] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu. Experiences of Landing Machine Learning onto Market-Scale Mobile Malware Detection. In *Proc. of ACM EuroSys*, pages 1–14, 2020.

[22] Google.com. Almost Native Graphics Layer Engine, 2021. https://github.com/google/angle.

[23] Google.com. Android Emulator: Simulates Android Devices on Your Computer, 2021. https://developer.android.com/studio/run/emulator.

[24] Google.com. Monkey: Automatic UI/Application Exerciser, 2021. https://developer.android.com/studio/test/monkey.

[25] Google.com. Stream Content with Chromecast, 2021. https://store.google.com/us/product/chromecast?hl=en-US.

[26] Google.com. SwiftShader: A CPU-Based Implementation of Graphics APIs, 2021. https://github.com/google/swiftshader.

[27] A. Herrera. NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation. *NVIDIA Corp*, pages 1–18, 2014.

[28] Huawei.com. Huawei's DevEco Studio, 2021. https://developer.harmonyos.com/en/develop/deveco-studio/.

[29] Intel.com. Houdini: Translate The ARM Binary Code into the x86 Instruction Set, 2021. https://www.intel.com/content/www/us/en/products/docs/workstations/resources/accelerate-game-development-houdini-optane-memory.html.

[30] Intel.com. Intel GVT-g: Full GPU Virtualization with Mediated Pass-through, 2021. https://github.com/intel/gvt-linux/wiki/GVTg_Setup_Guide.

[31] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. In *Proc. of ACM SIGCOMM*, pages 83–91, 2003.

[32] Kishonti Ltd. GFXBench: A Unified Graphics Benchmark Based on DXBenchmark, 2021. https://gfxbench.com/.

[33] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proc. of the Linux Symposium*, volume 1, pages 225–230, 2007.

[34] U. Laboratories. 3DMark: Popular Benchmarks for Gamers, Overclockers, and System Builders, 2021. https://www.3dmark.com/.

[35] LDPlayer.com. LDPlayer: Free Android Emulator for PC, 2021. https://www.ldplayer.net/.

[36] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proc. of ACM MobiSys*, pages 151–165, 2015.

[37] M. Li, H. Lin, C. Liu, Z. Li, F. Qian, Y. Liu, N. Sun, and T. Xu. Experience: Aging or Glitching? Why Does Android Stop Responding and What Can We Do About It? In *Proc. of ACM MobiCom*, pages 1–11, 2020.

[38] MEmu.com. MEmu: The Most Powerful Android Emulator, 2021. https://www.memuplay.com/.

[39] Mesa.org. The Mesa 3D Graphics Library, 2021. https://www.mesa3d.org/.

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

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

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

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

[44] J. Oberheide and C. Miller. Dissecting The Android Bouncer. *SummerCon2012, New York*, 95:110, 2012.

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

[46] QEMU.org. QEMU: A Generic and Open Source Machine Emulator and Virtualizer, 2021. https://www.qemu.org/.

[47] RealVNC.com. VNC; Remote Desktop Access, 2021. https://www.realvnc.com/en/.

[48] S. Shi and C.-H. Hsu. A Survey of Interactive Remote Rendering Systems. *ACM Computing Surveys*, 47(4):1–29, 2015.

[49] Skia.org. Skia: 2D Graphics Rendering Library, 2021. https://skia.org/.

[50] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?

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

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

[53] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[54] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. of ACM MobiSys*, pages 168–179, 2019.

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