A Benchmarking Framework for Interactive 3D Applications in the Cloud

Tianyi Liu
The University of Texas at San Antonio
tianyi.liu@utsa.edu

Sen He
The University of Texas at San Antonio
sen.he@utsa.edu

Sunzhou Huang
The University of Texas at San Antonio
sunzhou.huang@utsa.edu

Danny Tsang
The University of Texas at San Antonio
danny.tsang@my.utsa.edu

Lingjia Tang
University of Michigan, Ann Arbor
lingjia@umich.edu

Jason Mars
University of Michigan, Ann Arbor
profmars@umich.edu

Wei Wang
The University of Texas at San Antonio
wei.wang@utsa.edu

***Abstract*—With the growing popularity of cloud gaming and cloud virtual reality (VR), interactive 3D applications have become a major class of workloads for the cloud. However, despite their growing importance, there is limited public research on how to design cloud systems to efficiently support these applications due to the lack of an open and reliable research infrastructure, including benchmarks and performance analysis tools. The challenges of generating human-like inputs under various system/application nondeterminism and dissecting the performance of complex graphics systems make it very difficult to design such an infrastructure. In this paper, we present the design of a novel research infrastructure, *Pictor*, for cloud 3D applications and systems. Pictor employs AI to mimic human interactions with complex 3D applications. It can also track the processing of user inputs to provide in-depth performance measurements for the complex software and hardware stack used for cloud 3D-graphics rendering. With Pictor, we designed a benchmark suite with six interactive 3D applications. Performance analyses were conducted with these benchmarks, which show that cloud system designs, including both system software and hardware designs, are crucial to the performance of cloud 3D applications. The analyses also show that energy consumption can be reduced by at least 37% when two 3D applications share a could server. To demonstrate the effectiveness of Pictor, we also implemented two optimizations to address two performance bottlenecks discovered in a state-of-the-art cloud 3D-graphics rendering system. These two optimizations improved the frame rate by 57.7% on average.**

***Index Terms*—Cloud Computing, Cloud Gaming, Cloud Gaming Benchmarks, Cloud Gaming Performance Analysis, Cloud Graphics Systems**

I. Introduction

The rise of cloud gaming and cloud virtual reality (VR) has made interactive 3D applications a major class of workloads for cloud computing and data centers [6], [23], [37], [52], [56]. A main benefit of rendering user-interactive 3D applications in the cloud is that it may reduce the installation and operational costs for the large-scale deployments of these 3D applications. Running these 3D applications in the cloud also allows mobile clients with less powerful GPUs to enjoy better visual effects. Moreover, cloud 3D-graphics rendering may also simplify the development and delivery of these 3D applications. For the rest of this paper, we refer to these user-interactive 3D applications simply as *3D applications*.

Most prior research on virtual desktop infrastructure (VDI) or cloud gaming focused on network latency [2], [20], [41]. However, the network latency is considerably reduced today and becomes viable for cloud 3D applications [77]. This improved network, in turn, makes the design of *cloud 3D-graphics rendering systems* crucial to the efficiency and performance of cloud 3D applications. However, there is limited public research on this system design, largely due to the lack of an open and reliable research infrastructure, including benchmarks and performance analysis tools. This lack of research infrastructure even affects non-cloud VR system research, which employed non-uniform evaluation methodologies [9], [42], [84]. Prior attempts to provide such research infrastructures [12], [54], [85] have limited success due to the following challenges.

First, for reliable evaluation, the research infrastructure must be able to mimic human interactions with 3D applications under randomly generated/placed objects and varying network latency. That is, the inputs used for the benchmarks should closely resemble real human inputs, so that the performance results obtained with these human-like inputs are similar to those obtained with real human inputs. Prior research generated human-like inputs from recorded human actions [75], [85]. However, this recording does not work for 3D VR applications and games, which have irregular and randomly placed/generated objects in their frames. Additionally, variations in network latency may affect when a particular object will be shown on the screen, further limiting the usefulness of the recorded actions [85].

Second, to reliably measure performance, the research infrastructure must be able to accurately measure the round-trip time/latency to respond to a user input [54], which in turn, relies on the accurate association of a user input and

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00076

its response frame. However, this association is very difficult due to the need to track the handling of a user input and the rending of its response frame across the network, across the CPU and GPU, and across multiple software processes (i.e., tracking from the client to the server, and back to the client).

Third, to effectively identify performance bottlenecks, the research framework must be able to measure the performance of every stage involved in the handling of a user input and the rendering of its response frame. The framework should be able to properly measure the performance of all the components involved, including those from the complex graphics software stack and heterogeneous hardware devices. Additionally, the research framework must have low overhead to ensure these measurements are reliable.

Fourth, the research infrastructure should be extensible to easily include new 3D applications. 3D applications are typically refreshed every one or two years and most of them are proprietary. Therefore, the research infrastructure should be constantly refreshed with new 3D benchmarks without requiring to modify their source code.

In this paper, we present a novel benchmarking framework, called *Pictor*, which overcomes the above challenges to allow reliable and effective performance evaluation of 3D applications and cloud graphics rendering systems. Pictor has two components: 1) an intelligent client framework that can generate human-like inputs to interact with 3D applications; and 2) a performance analysis framework that provides reliable input tracking and performance measurements. Inspired by autonomous driving, the intelligent client framework employs computer vision (CV) and recurrent neural network (RNN) to simulate human actions [39], [72], [74]. The performance analysis framework tracks inputs with tags and combines various performance monitoring techniques to measure the processing latency and resource usage of each hardware and software component. Additionally, Pictor is carefully designed to have low overhead and requires no modification to 3D applications.

With Pictor, we designed a benchmark suite with four computer games and two VR applications. Through experimental evaluation with these benchmarks, we show that Pictor can indeed accurately mimic human action with an average error of 1.6%. We also conducted an extensive performance analysis on a state-of-the-art cloud 3D-graphics rendering system [67], [82] to characterize the 3D benchmarks and the rendering system, analyze the impact of co-locating multiple 3D applications, and study the overhead of rendering 3D applications in containers. This performance analysis demonstrated that executing 3D application in the cloud could provide good Quality-of-Service (QoS). The analysis also showed that cloud/server performance can be a major limitation on QoS. Hence, the design of the cloud system, including both software and hardware, is crucial to the performance of cloud 3D applications. Moreover, executing two 3D applications in the same cloud server might reduce energy consumption by at least 37% while still achieving good QoS. The performance analysis also showed that container-based virtualization incurred less than 2% overhead. At last, to demonstrate that the in-depth performance analysis allowed by Pictor can indeed lead to performance improvements, we implemented two optimizations which improved average frame-rate by 57.7%.

The contributions of this paper include:

1. A novel intelligent client framework that can faithfully mimic human interactions with complex 3D applications with randomly generated/placed objects and under varying network latency.

2. A novel performance analysis framework that can accurately track the processing of user inputs, and measure the performance of each step involved in the processing of user inputs in various software and (heterogeneous) hardware components. This framework is also carefully designed to have low overhead and requires no application source code.

3. A comprehensive performance analysis of a state-of-the-art cloud graphics rendering system, the 3D benchmarks and containerization. This analysis also shows the benefit of cloud 3D applications and reveals new optimization opportunities.

4. Two new optimizations for current cloud graphics rendering system with significant performance improvements, which also demonstrate the effectiveness of Pictor.

The rest of this paper is organized as follows: Section II discusses a typical cloud graphics rendering system; Section III presents the design of Pictor; Section IV evaluates the accuracy and overhead of Pictor; Section V provides the performance analysis on a current cloud graphics rendering system; Section VI presents two new optimizations; Section VII discusses related work and Section VIII concludes the paper.

II. Cloud 3D Rendering System

Figure 1 illustrates the typical system architecture for cloud graphics rendering. This architecture employs a server-client model where the servers on the cloud execute 3D applications and serve most of their rendering requests. The client is mainly responsible for displaying UI frames and capturing user inputs. The client may also perform less-intensive graphics rendering, depending on the system design. In this work, we focus on Linux-based systems and open-source software which are easy to modify and free to distribute.

The system in Figure 1 operates in the following steps. When the client's interactive device captures a user input (e.g., a keystroke, mouse movement, or head motion), it sends the input through the network to a proxy on the cloud server (step ❶), which forwards the input to the application (step ❷). The proxy is usually a server application that handles media communication protocols, such as a Virtual Network Computing (VNC) server with Remote Frame Buffer (RFB) protocol or a video streaming server with extended Real-Time Streaming Protocol (RTSP) [31], [67], [70]. After receiving the input, the application starts frame rendering (step ❸). A 3D application may use a rendering engine that provides functions for drawing complex objects (e.g., the "Application 1"), or it may directly call a 2D/3D library to draw objects from scratch (e.g., "App 2"). The rendering engine, in turn, invokes the 2D/3D library. On Linux, the 2D/3D library is typically

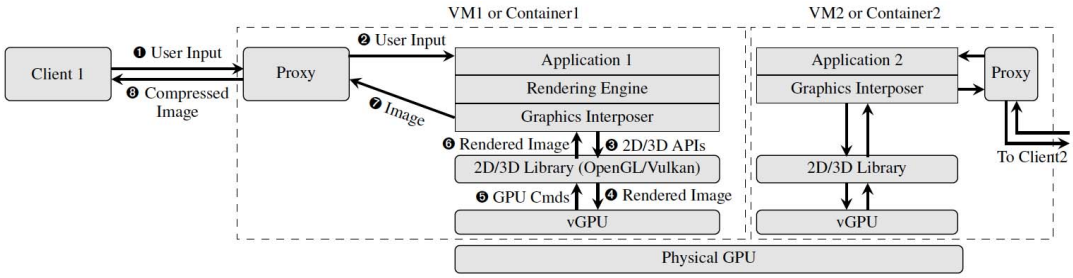


Fig. 1. System architecture for cloud graphics rendering.

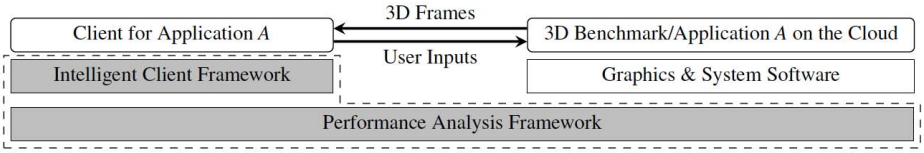


Fig. 2. Overview of the Pictor benchmarking framework (surrounded by the dashed box).

Mesa 3D Graphics Library, which implements the APIs of OpenGL and Vulkan [32], [62], [71]. Examples of the rendering engine include Unity, OSVR, and OpenVR [13], [73], [78]. To ensure 2D/3D calls are indeed invoked on the server, a graphics interposer library is employed [17]. The 2D/3D library (and the GPU driver) then translates the drawing APIs into GPU commands to perform the rendering on the GPU (step ❹). After the frame is rendered on the GPU, the graphics interposer copies the frame from the GPU (step ❺-❻) and push the newly rendered frame to the server proxy (step ❼). The proxy then compresses and sends the frame over the network to the client for display (step ❽).

Moreover, as shown in Figure 1, multiple 3D applications can execute simultaneously on the same machine and share hardware components, such as CPU, memory, GPU, and PCIe buses. Each application is executed in a virtual machine (VM) or a container with virtualized GPUs (vGPU) [25], [40], [57].

This system architecture has two implications for the benchmarking of cloud graphics rendering systems. First, as the behaviors of 3D applications are heavily influenced by user inputs, reliably benchmarking 3D applications requires generating human-like inputs. Second, cloud graphics rendering system includes complex and heterogeneous software/hardware components, which must be properly handled/measured when analyzing performance. In the rest of this paper, we will describe the design of Pictor, which overcomes the challenges mention in Section I and the above two issues.

III. The Design of Pictor

Figure 2 shows the components of Pictor benchmarking framework. A main component of Pictor is the intelligent client framework that is used to generate clients with human-like actions to interact with 3D applications. The other main component of Pictor is the performance analysis framework, which spans over the client and the server to provide reliable performance measurements. The rest of this section describes these two components in detail.

A. Intelligent Client Framework Design

Overview The intelligent client framework allows building an intelligent client for a 3D application by learning how to properly interact with this application from recorded human actions. More specifically, for a 3D application, an RNN model is trained based on recorded human actions under a scene of this application [72]. To improve the RNN model's accuracy, the objects in the frames are first recognized using computer vision (CV) with Convolutional Neural Network (CNN) [39].

Figure 3 gives an overview of a client obtained with the intelligent client framework, which operates in the following steps. After a compressed frame is sent over the network to the intelligent client (step ❶), it is first decompressed (step ❷). The decompressed frame is then processed by a CNN model to recognize its objects (step ❸). The types and coordinates of the recognized objects are then sent to an RNN model to generate user inputs that mimic real human actions (step ❹).

These inputs are eventually sent back to the client proxy, which encodes these actions into network packages and sends them to the benchmark (step ❺). With CNN and RNN models, the clients can properly interact with 3D applications with random frames and under random networking/system latency. By generating actions purely based on frames, the clients can be built for 3D applications without knowing their internal designs or modifying these applications. Note that, for some simple 3D applications, instead of RNN, simple rule-based input generation may suffice. Nonetheless, RNN provides a generic solution that works well with any 3D application.

Model Training Each 3D application/benchmark has its own CNN/RNN models, which are trained from a recorded session of human actions under an application scene. The intelligent client framework provides tools to perform this recording. Each recorded session includes a sequence of frames and the corresponding human actions to each frame. To train a CNN model, the objects in the frames need to be manually labeled. The labeled frames are then fed into a machine-learning (ML)

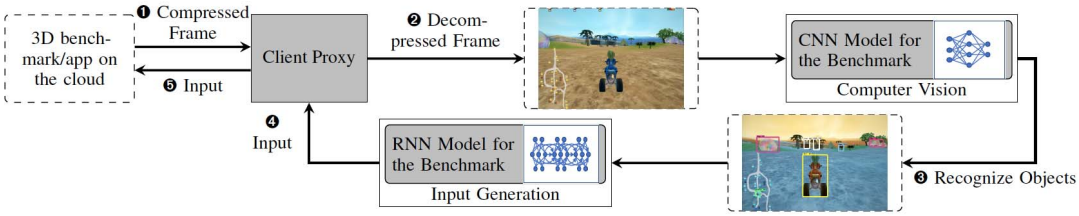


Fig. 3. Overview of the intelligent client. The image is obtained from a racing game, SuperTuxKart [28].

package to train the CNN model. The manual frame labeling is generally fast and takes about 4 hours for one 3D application in our experience, as only the objects that can determine the user inputs need to be labeled.

An RNN model can also be trained using the recorded session. The recorded frames are first processed by its CNN model to recognize the objects. After the recognition, the recorded data are converted into a training data set where the features are the objects in a frame and the labels are the corresponding human actions. An RNN model can then be trained to learn how to respond to the objects in a frame like a real human. Note that, our goal is not to train an AI to compete with human. Instead, we aim at training an RNN model to mimic human actions under varying system latency and frame randomness, so that the performance results obtained with the RNN-generated actions are similar to those obtained with real human users. Because a trained RNN model is executed on the same scene where it is trained, the model is likely to work well as long as it has low training loss.

Implementation We implemented the training and inference of the CNN and RNN models with Tensorflow [1]. The actual CNN model used is MobileNets [30]. The actual RNN model used is Long Short-Term Memory (LSTM) [29].

B. Performance Analysis Framework Design

Overview The performance analysis framework provides performance measurements of 3D applications and cloud graphics rendering systems. Performance measurements include frame rate (FPS, frames-per-second), the latencies of each stage involved in the handling of a user input and the delivery of its response frame, as well as system-level and architecture-level resource usages. As stated in Section I, designing this framework has two difficulties. The first difficulty is to accurately track and associate the processing of an user input and the rendering of its response frame. The second difficulty is to measure the performance of the complex and heterogeneous software/hardware components. This section describes how Pictor overcomes these difficulties.

Tracking User Input Processing. To track the input processing, we tag the input from the client and use the tag to identify every stage of the input processing. More specifically, at the beginning of each processing stage, the tag of the corresponding user input is extracted from the input data. At the end of the stage, the tag is added to the output data, allowing the next stage to extract it. For graphics rendering, the start and end of each stage can be determined based on the invocation of specific OpenGL and X-Window APIs, and the tags can be passed along as input/output data to these APIs. The invocations to these APIs can be intercepted with API hooks, allowing extracting/adding the tags in these hooks.

TABLE I
SOME OF THE APIS INTERCEPTED AT THE API HOOKS.

Hooks	Intercepted APIs
$Hook_4$	XNextEvent, glutKeyboardFunc
$Hook_5$	glxSwapBuffer, glutSwapBuffers
$Hook_6$	glReadBuffer, glReadPixel
$Hook_7$	XShmPutImage, glMapBuffer

Figure 4 illustrates the API-hook-based input-tracking technique. For now, we assume a sequential graphics rendering process. To track an input, $hook_1$ at the client proxy gives every input a unique tag and sends the tag with the input to the server proxy. Upon receiving the input, $hook_2$ at the server proxy extracts the tag from the network package. The tag is then forwarded to the application with its input by $hook_3$. When the application receives the input, the tag is extracted at $hook_4$ and saved. $Hook_5$ marks the start of the GPU rendering, there is no need to send the tag to GPU. At $hook_6$, the saved tag is embedded into the pixels of the rendered frame (the old pixels are stored in shared memory). Embedding the tag in pixels ensures that the tag survives the inter-process communications between the application and server proxy. After the server proxy receives the tagged frame at $hook_8$, it extracts the tag, restores the modified pixels, and sends the frame with the tag to the client. Once $hook_{10}$ at the client proxy receives the tagged frame, it matches the tag with a previously sent user input, which finishes the tracking. Table I gives some examples of the APIs that can be intercepted from $hook_4$ to $hook_7$. The other hooks in the server and client proxies can be easily identified using their source code.

However, instead of the above sequential rendering process, modern graphics applications typically employ software pipelines to parallelize the rendering for better performance. Figure 5 shows the typical stages of this pipeline for remote 3D-graphics rendering when rendering two frames, $frame_i$ and $frame_{i+1}$. As Figure 5 shows, in each pass of the pipeline, a new frame is rendered, and the previous frame is copied and sent to the clients. For example, in the first row of Figure 5, $frame_i$ is rendered based on $input_i$, while $frame_{i-1}$ is copied from the GPU and sent to the client.

Note that, Figure 5 shows the pipeline for the cloud rendering system analyzed in Section V. In this system, the stages of

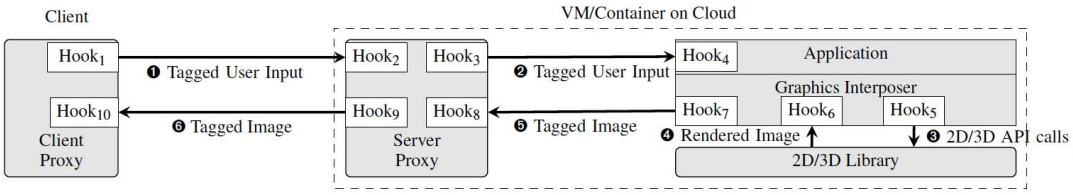
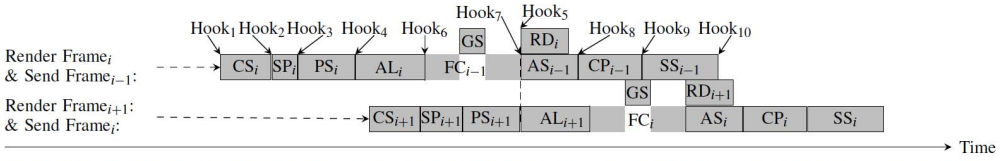


Fig. 4. Using API hooks to track the processing of a user input and the rendering of its response frame.



Stages: **CS**$_i$: client sends input$_i$ to server. **SP**$_i$: server proxy receives input$_i$. **PS**$_i$: server proxy sends input$_i$ to app. **AL**$_i$: app logic for frame$_i$ on CPU. **FC**$_i$: frame$_i$ copied from GPU to CPU. **AS**$_i$: app sends frame$_i$ to server proxy. **CF**$_i$: compress frame$_i$ on server proxy. **SS**$_i$: server send frame$_i$ to client. **GS**: GPU sends frame to application/graphics-interposer **RD**$_i$: rendering frame$_x$ on GPU.

Fig. 5. A typical software pipeline for open-source remote 3D-graphics rendering on Linux.

application-logic (AL) and frame-copy (FC) are carried out by the same thread due to the difficulty to know when a frame is completely rendered in the GPU. Therefore, AL and FC stages cannot overlap, and the next AL stage must start after the previous FC stage is finished. Nonetheless, any other two stages in this pipeline can overlap, as they are not carried out by the same thread/processor.

The main benefit of this software pipelining is that it allows the CPU and GPU to execute simultaneously. For example, as shown in Figure 5, when frame$_i$ is being rendered on the GPU (stage RD$_i$), the CPU is working on the application logic for frame$_{i+1}$ (stage AL$_{i+1}$) and sending frame$_{i-1}$ (stage AS$_{i-1}$) using two threads/cores. The tag-based input tracking still works for this parallel rendering, as long as the tracking implementation is aware that the processing/rendering of an input spans over two passes of the pipeline.

Performance Measurements for Diverse Components. The API hooks also allow measuring the execution times (latencies) of each stage involved in the rendering. A hook records a timestamp when it intercepts an API call. The differences between the timestamps of two hooks then give the time spent in each stage. For example, the time difference between the hook$_{10}$ and hook$_1$ with matching tag gives the round-trip time (RTT) to handle a user input.

However, the time measured with the hook's CPU timestamps cannot give GPU processing time. To obtain GPU time, we use the time-querying functionality of OpenGL [24]. Start and stop querying statements are inserted into the hooks to measure the GPU time spent in each stage. For example, the time query starts at a hook$_5$ and ends at the subsequently-invoked hook$_6$ gives the GPU time to render a frame.

Pictor also measures FPS and resource usages. The FPS is obtained by counting the frames at the server and client proxies. System-level resource usages, such as CPU/GPU and memory utilizations, are obtained from the OS and GPU drivers [8], [58]. Architecture-level resource utilization is measured using hardware performance monitoring units (PMU). CPU PMU readings for each stage are obtained by using PAPI inside the API hooks [79]. The PMUs on AMD GPUs are queried using AMD's GPU Performance API [7]. For NVidia GPU, an external tool, NSight Graphics, is used to read PMUs, as NVidia does not support programmable PMUs reading for graphics rendering on Linux.

TABLE II
APPLICATIONS INCLUDED IN OUR BENCHMARK SUITE.

Application Area	Benchmark
Game: Racing	SuperTuxKart (STK) [28]
Game: Real-time Strategy	0 A.D. (0AD) [22]
Game: First-person Shoot	Red Eclipse (RE) [65]
Game: Online Battle Arena	DoTA2 (D2) [83]
VR: Education/Game	InMind (IM) [55]
VR: Heatlh	IMHOTEP (ITP) [63]

Performance Measurement Extensibility and Overhead. One benefit of using API hooks is that it does not require modifying 3D applications. Our performance analysis framework can be applied to any proprietary 3D applications, as long as these applications invoke standard 3D APIs, such as those given in Table I. As later shown in the experimental evaluation (Section IV), these API hooks also incur little overhead. However, the time queries used to measure the GPU performance may stall the CPU and thus incur a high overhead. To mitigate the impact of these stalls, we used two query buffers and switched them between frames.

C. The Benchmark Suite

With Pictor, we designed a benchmark suite, which contains four computer games and two VR applications. All benchmarks are from real applications and cover popular game genres and usage cases. Table II lists these benchmarks. Among the six benchmarks, *Dota2* and *InMind* are closed source. Note that, as Pictor is designed to be extensible, new 3D applications can be easily added in the future.

IV. Evaluation

This section provides the experimental evaluation of the reliability/accuracy and overhead of Pictor.

Experiment Setup The benchmarks were executed on a server with an 8-core Intel i7-7820x CPU, 16GB memory, and an NVIDIA GTX1080Ti GPU with 11GB GPU memory. The clients consisted of four machines each with a 4-core Intel i5-7400 CPU and 8GB memory. The server and clients were connected using 1Gbps networks. 1Gbps network was chosen because it behaved similarly to 5G cellular networks in terms of the frame-transmitting latency as shown later in Section V-A2. Precision Time Protocol [21] was used to synchronize the time between the server and clients.

The server and clients run Ubuntu 16.04 as the OS and TurboVNC 2.1.90 [82] as the rendering system. We chose VNC as it has complete support for 3D rendering. The other open-source solution, GamingAnywhere [31], failed to run all of our benchmarks. To the best of our knowledge, all VNC implementations (and even the non-VNC proprietary NX technology [64]) required TurboVNC's graphics interposer, VirtualGL [17], to support 3D rendering. Therefore, we evaluated TurboVNC, as it represents the state-of-the-art open-source remote 3D rendering. We modified TurboVNC to support VR device inputs. All benchmarks were executed at a resolution of 1920×1080 with maximized visual effects.

Intelligent Client Accuracy Evaluation. To evaluate if the intelligent clients (ICs) indeed allow reliable and accurate performance results, we compared the benchmarks' behaviors under the ICs and human interactions. More specifically, each benchmark was executed using its IC and was also played/used by a real human user for three 15-minute sessions each (results were stable after 10 min). We then compared the performance results obtained from the two types of executions, including the latency, FPS, and CPU/GPU utilization. Figure 6 shows the round-trip time (RTT) that it took to process input for each benchmark when executed with the IC and the human user. For each execution, Figure 6 shows the mean, 1%-tile, 25%-tile, 75%-tile, and 99%-tile of the measured RTTs. As Figure 6 shows, the RTTs obtained with IC were very similar to those from the human. Table III also gives the percentage errors of the means of the RTTs obtained with our IC. The maximum percentage error for the mean-RTT for IC is only 2.5%, and average error for IC is only 1.6%, the RTTs from the IC and human runs were also similar. The data for other performance metrics were also similar for both runs. However, limited by space, other performance metric results are omitted.

Intelligent Client Speed Evaluation. Figure 7 gives the average times that it took to conduct CV (CNN) and generate input (RNN) for each benchmark. As the figure shows, the clients have fast inference times, with an overall average of 72.7ms for CV and 1.9ms for input generations. This fast inference allows the ICs to generate 804 actions per minute (APM) on average, which is faster than professional game players (about 250 to 300 APM) [50], [81], showing that the ICs can generate inputs fast enough to mimic human reaction speed. Note that, to ensure our ICs can faithfully mimic human actions, we slowed the rate of action generation to be around 250 APM.

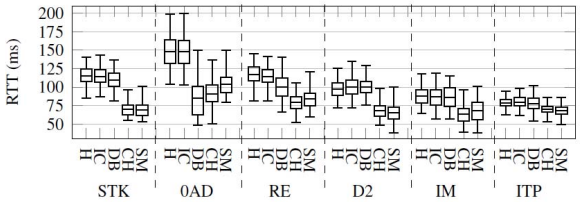


Fig. 6. The performance (RTT) distributions obtained with human users (H), Pictor's intelligent clients (IC), DeskBench [85] (DB), Chen, et al. [15] (CH), and Slow-Motion [54] (SM).

TABLE III
Percentage errors for the means of the RTTs obtained with Pictor's IC, DeskBench [85] (DB), Chen, et al. [15] (CH), and Slow-Motion [54] (SM), when compared to the mean RTTs obtained with human users.

	STK	0AD	RE	D2	IM	ITP	Avg
Pictor	0.8%	0.1%	2.5%	3.2%	1.3%	2.0%	1.6%
DB	5.4%	42.9%	14.6%	3.3%	1.3%	2.3%	11.6%
CH	39.5%	38.9%	32.6%	29.9%	11.4%	27.8%	30.0%
SM	39.8%	30.1%	28.2%	32.7%	13.7%	22.7%	27.9%

Pictor Overhead Evaluation. To evaluate the overhead of Pictor, we executed each benchmark with and without the performance analysis framework. For the run without the performance analysis framework, native TurboVNC is used with our ICs. As the native TurboVNC does not provide RTT readings, we compared the FPS of both runs. Our results show the performance analysis framework has low overhead. The FPS reduction was only 2.7% on average (5% at maximum) for all benchmarks. This low overhead is partially due to our use of double-buffers when querying GPU execution times. Without these double-buffers, the overhead was up to 10%.

Comparison with Prior Work. To show the importance of properly handling irregular/random objects and tracking inputs, we also compared Pictor with three prior performance measuring techniques for VDI and cloud gaming.

We first compared Pictor with DeskBench [66]. DeskBench was based on VNCPlay [85] and replayed recorded human actions to generate inputs. However, DeskBench did not only record an action, it also recorded the screen frame when this action was issued. During replay, the action was only issued when the displayed frame was similar to the recorded frame. With this frame comparison, DeskBench (and VNCPaly) only issued an action when the expected object was displayed, and thus, was capable to handle network latency variation.

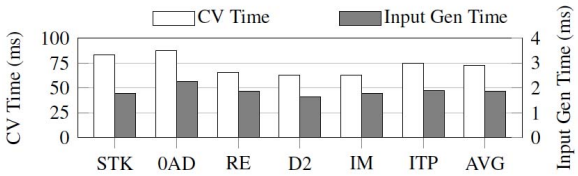


Fig. 7. Computer vision and input generation time.

Note that, the "similarity" between frames was a tune-able parameter for DeskBench. We tested with several parameter values following the methodology presented by DeskBench and reported the DeskBench's results using the best parameter we found. Additionally, as DeskBench did not provide input tracking, it was only used to generate inputs, and Pictor's performance framework was used to collect performance data. Figure 6 and Table III also give the RTT distributions and errors obtained with DeskBench. The average error of the mean-RTT obtained with DeskBench was 11.64%, which was considerably higher than the 1.6% error of Pictor. DeskBench was designed for 2D applications with well-shaped and placed objects (e.g., icons and texts), where simply comparing pixels can determine if an object is shown or not. However, for 3D games, even the same object can have different pixels and locations depending on the viewing angle and the flow of events. Hence, simply comparing the pixels is practically impossible to determine the existence of an object, causing DeskBench to frequently delay an action.

We also compared Pictor with a cloud gaming performance analysis methodology presented by Chen et al. [15]. In this methodology, the authors generated inputs with human players. This methodology did not provide input tracking, and hence, could not measure RTT at the client. Therefore, it had to compute the RTT by summing the time of the stages of CS, SP, AL, CP, and SS of the software pipeline. Figure 6 and Table III also give the RTT distributions and errors obtained with this methodology. The average error of the mean-RTT obtained with this methodology was 30.0%, which was also much higher than the 1.6% error of Pictor. There are two issues with this methodology because of the lack of input tracking. First, the AL latency in this methodology was obtained offline without the VNC server proxy. This offline measurement gave lower AL latency than that obtained during online execution, because it eliminated the resource contention between the game and the VNC server proxy. Second, with input tracking, the methodology could not measure the delays of the inter-process communication stages, including PS, FC, and AS. Because of these two issues, Chen et al.'s methodology usually reported smaller RTTs than those directly measured at the client.

The last comparison was conducted with a VDI performance measuring technique call Slow-Motion [12], [54]. Slow-Motion was designed to determine the RTT of one frame. Slow-Motion injected delays into the cloud rendering system to only allow one input/frame being processed at a time – only after an input was processed, its frame was rendered and sent to the client, could the processing of the next input/frame start. By allowing only one frame at a time, it was trivial to associate an input with its response frame. Note that, as Slow-Motion did not include an input generation technique, Pictor's IC was used to generate the inputs. Figure 6 also shows the RTT distributions obtained with Slow-Motion. The average error of the mean-RTT obtained with this methodology was 27.9%, which was also higher than the 1.6% error of Pictor. The main issue of Slow-Motion was that the injected delay changed the resource usage and the behavior of the benchmark and

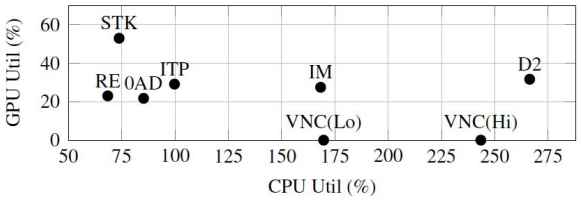


Fig. 8. CPU and GPU utilization for each benchmark.

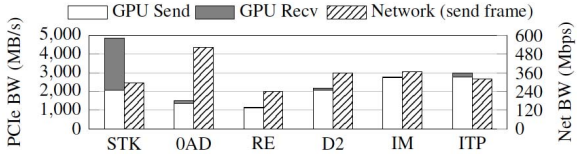


Fig. 9. Network and PCIe (send-to and received-at the GPU) bandwidth usages for each benchmark.

the VNC server proxy (which was also noted by the original authors [54]). Because only one frame was rendered at a time, the resource contention caused by parallel processing/rendering of the inputs and frames was eliminated, and the resource contention between the benchmark and the server proxy was also reduced. Consequently, Slow-Motion typically reported smaller RTTs than those observed with a system executing at full capacity.

V. Performance Analysis of Cloud Rendering System and 3D Applications

A. Perf. Analysis with A Single Benchmark

This section provides the performance analysis results with a single benchmark, which was executed using the same methodology given in Section IV.

1) System-level Resource Utilization: Figure 8 gives the CPU and GPU utilization of each benchmark. The CPU utilization of these benchmarks had a high variation, ranging from 68% (*RedEclipse*) to 266% (*Dota2*). The GPU utilization also had a high variation, ranging from 22% to 53%. The VNC server also had considerable CPU utilization, which varied from 169% to 243%, depending on the FPS and frame compression difficulty. The CPU memory usages also vary considerably, ranging from 600MB (*Dota2*) to nearly 4GB (*InMind*). The GPU memory usages of these benchmarks were less than 800MB, which is similar to the 1GB-2GB GPU memory requirements of recent popular games.

Figure 9 shows the network and PCIe bandwidth usages for each benchmark. For network usage, only the bandwidth usage of sending the frames to the client is shown, as sending the inputs from the clients used only 1.5Mpbs. The network usages of these benchmarks were below 600Mpbs, which is lower than the maximum bandwidth of the coming 5G cellular network and 10Gpbs broadband. Similarly, all benchmarks used less than 5GB/s on the PCIe bus, which is well below the 31.5GB/s maximum bandwidth of PCIe3. Except for *SuperTuxKart*, all benchmarks sent limited amount of data from the CPU to GPU,

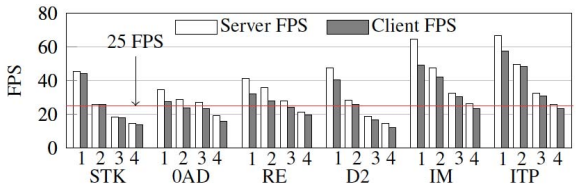


Fig. 10. Average server and client FPS when executing one to four instances of the same benchmark on the server.

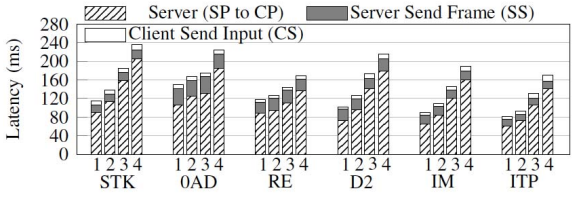


Fig. 11. RTT breakdown when executing one to four instances of the same benchmark on the server.

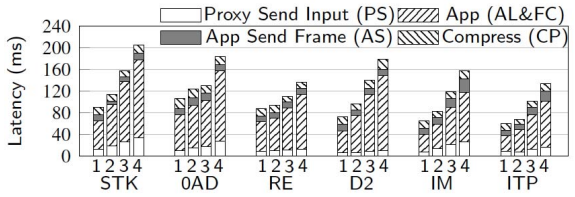


Fig. 12. Server time breakdown when executing one to four instances of the same benchmark on the server.

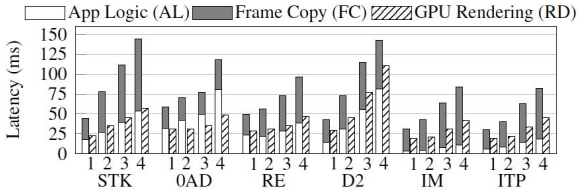


Fig. 13. Application time breakdown when executing one to four instances of a benchmark on a server.

suggesting most of their rendering data were stored on the GPU. The exception of *SuperTuxKart* was likely due to its frequent and drastic changes in the rendered frames. For all benchmarks, there is high PCIe bandwidth usage from GPU to CPU, which represented the data used for copying rendered frames from GPU to CPU.

2) Application Performance: Figure 10 gives the server and client FPS for each benchmark when one to four instances of the same benchmark was executed on the server. Here, we focus on the FPS for one instance (i.e., the bars with x-axis label "1"). Server FPS measured the number of frames that were generated at the server in one second. Client FPS measured the number of frames the client received in one second. The lowest client FPS was 27 (for *0AD*), which is still higher than the minimum 25 FPS quality-of-service (QoS) requirement for 3D applications, showing the feasibility of cloud graphics rendering [68]. Note that, Figure 10 shows the average server/client FPS for each benchmark. Nonetheless, the lowest observed FPS was still higher than 25.

Figure 11 gives the average RTTs of handling an input for each benchmark. Again, we focus on the RTTs for one instance (i.e., the bars with x-axis label "1"). These RTTs are broken down into the time the sever spent on handling the input and the network times for sending the inputs and frames. For all benchmarks, the network latency for sending inputs (stage CS) was very small (¡ 10ms). The network latency for sending frames (stage SS) ranged from 14ms to 35ms, which was similar to those reported by prior work with 4G/5G cellular network, suggesting our 1Gpbs network is close to the real use case [77]. The largest component of RTT was always the time that the server took to process inputs, which include all stages from SP to CP. This server processing time ranged from 61ms to 106ms. Such high server time indicates that cloud system design is crucial to ensure good performance.

In Figure 12, the server time is further broken down into the time of VNC sending inputs to the benchmark (stage PS), the application execution time (stage AL, FC, and RD), benchmark sending frame to VNC (stage AS) and the time of VNC compressing frames (stage CP). Note that, the time for stage SP is omitted because it was too small ($< 1ms$) to be visible in the figure. As Figure 12 shows, for all benchmarks, the main component of the server processing time is the application execution time, where the execution times for other stages (i.e, PS, AS, and CP) were less than 18ms.

The application execution time was further broken down in Figure 13. As GPU rendering (RD) executes in parallel with the application logic (AL) and frame copy (FC) stages, the GPU rendering times are shown as separate bars in Figure 13. Surprisingly, many benchmarks spent most of their time on copying frames. This long frame-copy time was due to the long PCIe transporting time and inefficiency implementation, which were addressed with new optimizations in Section VI. Moreover, because of the long frame-copy, GPU rendering was never the performance bottleneck in our experiments.

3) Architecture-level Resource Usages: Figure 14 shows the CPU cycles for each benchmark using the Top-Down analysis [35]. The CPU cycles are broken down into front-end stalls, back-end stalls, bad speculation stalls, and the cycles for instruction retirements. As Figure 14 shows, all benchmarks had long back-end stalls and low instructions-per-cycle, indicating

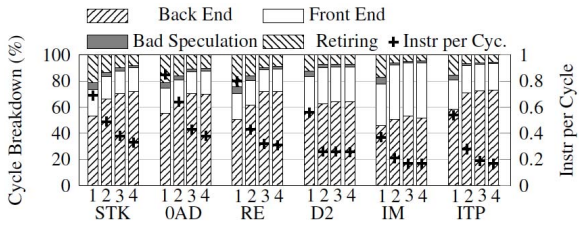


Fig. 14. CPU cycles breakdown when executing one to four instances of a benchmark on the same server.

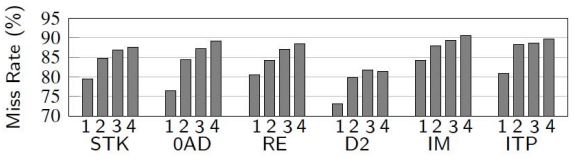


Fig. 15. L3 cache miss rates when executing one to four instances of a benchmark on the same server.

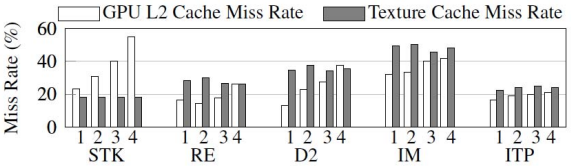


Fig. 16. GPU L2 and texture cache miss rates when executing one to four instances of a benchmark.

these benchmarks were likely memory-bound. As their L3 cache miss rates (L3-misses/L3-accesses) were also very high ($> 70\%$) (Figure 15), it can be deduced that these benchmarks are also off-chip memory bound. This behavior is consistent with typical graphics rendering implementation where uncached memory is used for CPU-to-GPU communications [34].

As shown in Figure 16, all benchmarks except *InMind* had moderate GPU cache miss rates. These moderate cache miss rates suggested most benchmarks can use GPU caches relatively effectively. Note that, *0AD* used OpenGL v1.3, which is not supported by NVidia PMU reading tools. Therefore, we could not obtain GPU cache miss rates for *0AD*.

4) Single Benchmark Analysis Summary: 1) Executing 3D applications in the cloud can provide reasonable QoS with current hardware and network. 2) Cloud/server performance can be a major limitation on FPS and RTT. Therefore, optimizing the cloud system design is crucial for cloud graphics rendering. 3) 3D applications have a wide range of resource demands and behaviors, suggesting cloud system optimizations may need to consider individual application's characteristics. 4) 3D applications intensively utilized the CPU, GPU, memory, and PCIe buses. Consequently, cloud system optimizations need to consider the impacts of all these resources. For instance, we designed an optimization to handle the long frame-copy time over the PCIe bus in Section VI.

B. Perf. Analysis with Multiple Benchmarks

To investigate the feasibility and analyze the performance of multiple 3D applications sharing hardware in the cloud, we also conducted experiments with multiple 3D benchmarks. More specifically, we executed one to four instances of the same benchmark on our server. Each benchmark instance interacted with its own client machine. To ensure enough network bandwidth, each benchmark instance used its own 1Gpbs network card on the server.

1) Server Power Consumption: We obtained the server power consumption using a Klein Tools CL110 meter. Overall, adding a new instance only increased the total server power

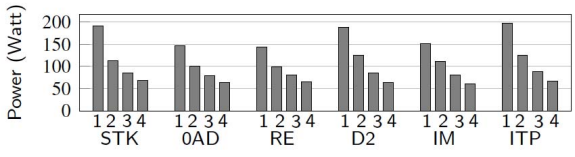


Fig. 17. Per-instance power usage when executing one to four benchmark instances on one machine.

consumption by less than 20%. As shown in Figure 17, this small increase in total power usage translated into per-instance power usage reductions of 33%, 50%, and 61%, when running two to four instances (comparing to one instance). These power usage reductions demonstrate a main benefit of executing 3D applications in the cloud – the reduced energy cost and operational cost.

2) Application Performance: Figure 10 also shows the FPS for each benchmark when two to four instances of the same benchmark were executed on the same server. Here, we focus on the FPS for 2 to 4 instances (i.e., the bars with x-axis labels "2", "3" and "4"). As Figure 10 shows, for all benchmarks, executing with two instances could still provide an acceptable (i.e., $\geq$ 25) FPS. For three benchmarks, *RE*, *IM*, and *ITP*, executing three instances could still achieve an FPS higher than 25. These FPS results show that consolidating multiple 3D applications on one server can still provide acceptable QoS, and thus reduce infrastructure cost.

Figure 11, 12, and 13 also give the breakdown of the RTT, server processing time, and benchmark processing time, when two to four instances of the same benchmark were executed on the same server. Again, we focus on the results for 2 to 4 instances (i.e., the bars with x-axis labels "2", "3" and "4"). As Figure 11 shows, there was no significant increase in network time due to the use of multiple graphics.

However, there were significant increases in server execution times. As shown in Figure 12 and Figure 13, nearly every execution stage on the server experienced increased execution time with more instances. We have observed high increase (up to 96%) in execution time for the stages with inter-process communications (IPC), including the stages of PS and AS. There were also significantly increase in the stages that do computations on the CPU and GPU, including the stages of AL, FC, RD, and CP. In particular, the average application logic (stage AL) time increased by 235% when executing with four instances, and the average GPU rendering (stage RD) time increased by 133% when executing with four instances. These increased execution time on CPU and GPU were main caused by two issues – the oversubscribed CPU/GPU when executing three or four instances, and the hardware resource contention that happened in the CPU, GPU, and PCIe buses. This contention is discussed in detail in the following section.

3) Architecture-level Resource Usages: Figure 14 and Figure 15 gives the CPU cycle breakdown and the L3 cache miss rates of one benchmark instance, when it was executed with other benchmark instances. As the figures show, both the back-end stalls and the L3 miss rates increased considerably with

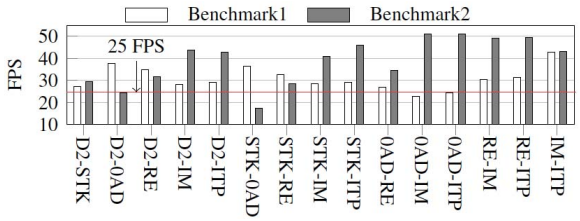


Fig. 18. Client FPS for 15 pairs of benchmarks.

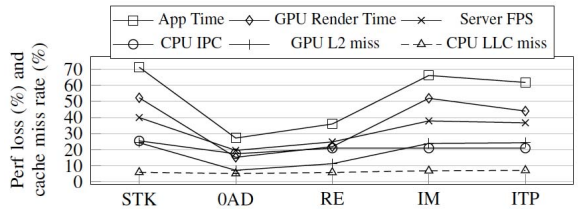


Fig. 19. Performance loss and cache miss increases of *Dota2* when executing with other benchmarks. Higher values indicate higher loss and contention.

more benchmark instances, indicating that there was heavy contention in the memory system.

Memory contention was also observed in the GPU. As shown in Figure 16, all benchmarks experienced increased GPU L2 miss rates, which contributed to the increase in their GPU rendering time. This L2 miss increases may be explained with the GPU internal graphics pipelines [48]. Because of this pipeline, there may be frames from different benchmark instances rendered simultaneously, thus causing the L2 cache contention. The texture cache miss rates, however, did not change significantly, as it is a private cache. Note that, for both CPU and GPU, the contention may exist beyond cache and extend to DRAM and PCIe buses. Resource contention also existed between the benchmarks and VNC proxies. However, a full contention analysis is beyond the scope of this paper and will be conducted in the future.

4) Multi-benchmark Analysis Summary: 1) Executing multiple 3D applications on the same server in the cloud can provide acceptable QoS while significantly reduced energy consumption. Therefore, cloud graphics rendering may considerably reduce the infrastructure and operational costs for the large-scale deployment of 3D applications. 2) Resource contention and slowed IPC can severely degrade the performance of 3D applications in the cloud, and thus should be properly managed. Moreover, resource contention simultaneously exists in the CPU and GPU (and potentially in the PCIe buses). Contention also exists between the applications and the server proxies. Therefore, resource contention and IPC management for cloud graphics rendering should be designed with heterogeneity in mind.

C. Perf. Analysis with Mixed Benchmarks

To study the impact of colocating different 3D applications, we also conducted experiments where two different benchmarks were executed simultaneously. As there were 6 benchmarks, a total of 15 pairs of them were evaluated.

1) Application Performance, Power Consumption, and Architectural-level Resource Usages: Figure 18 gives the client FPS of the 15 pairs of benchmarks. Server FPS was just slightly higher than client FPS and was omitted due to space limitation. As Figure 18 shows, 11 pairs of benchmarks had client FPS higher than 25, suggesting different 3D applications can also share hardware while ensuring acceptable QoS. We also observed that adding an additional benchmark only increase the total server energy consumption by no more than 25%. Therefore, comparing to running two applications on two servers, executing two different 3D applications on the same server can reduce energy consumption by at least 37%.

Similar to the observations in Section V-B, oversubscribed CPU/GPU, prolonged IPC, and resource contention significantly increased the server execution time. Nonetheless, we also observe that the contentiousness of these benchmarks varies considerably. Figure 19 gives the performance and CPU/GPU cache misses of *Dota2* when it was executed with different benchmarks. Other benchmark pairs showed similar results and were omitted due to space limitation. As Figure 19 shows, there was significant variation in *Dota2*'s performance depending on its co-runners, with *SuperTuxKart* causing the highest contention and *0AD* causing the least contention. This high variation in the contentiousness may be utilized in optimizations (e.g., selecting the proper set of 3D applications to share hardware). It is also interesting to observe that the contentiousness for CPU cache and GPU cache seemed to have high correlation. This correlation may be due to the rendering data being shared between CPU and GPU, and may be exploited when managing 3D applications contention (e.g., predicting a 3D application's contentiousness).

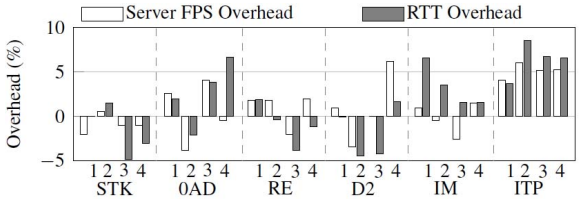


Fig. 20. Server FPS/RTT overheads of containers. Negative overheads are performance improvements.

2) Mixed Benchmark Analysis Summary: 1) Executing multiple different 3D applications on the same server in the cloud can provide acceptable QoS while significantly reducing energy consumption. 2) The contentiousness of 3D applications varies considerably, which may be utilized in system optimizations. 3) There may also be a correlation between the contentiousness for the CPU cache and GPU cache, which may also be exploited in system optimizations.

D. Container Overhead

So far, all experiments were conducted on bare-metal systems. However, as we target cloud computing, it is also important to analyze the performance impact of virtualization and

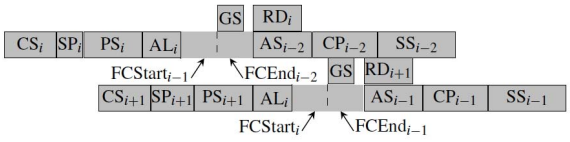


Fig. 21. Optimizing frame copy with two-step copy.

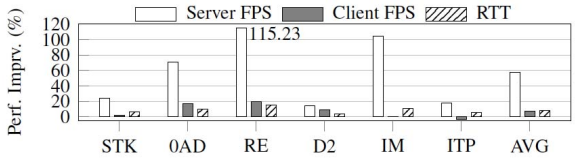


Fig. 22. Improved FPS/RTT with our optimizations.

containerization. Hence, we repeated the above the performance analysis experiments using Docker containers to study the overhead of containerization. More specifically, we executed each instance of the benchmark and its VNC server inside an NVidia Docker container [57]. We chose container instead of VM because docker container currently supports most GPUs, while VM-based virtualization requires special GPUs. We will evaluate VM-based systems in the future.

Figure 20 shows the percentages of reduced FPS and increased RTT (i.e., FPS and RTT overheads), comparing to the runs without virtualization. On average, Docker containers incurred little overhead. The average overhead for RTT was only 1.3%, and the average overhead for server FPS was only 1.5%. This low average overhead further shows the feasibility of executing 3D applications in the cloud.

Nonetheless, the overhead can still be as high as 8.5% for RTT (or 6% for FPS). We observed that these overheads were usually due to increased execution time for the stages with IPC (stages PS and AS). These high overheads show the need to optimize containerized cloud 3D applications to ensure that worst-case performance still meets QoS goals. Besides the overheads for RTT and FPS, the GPU rendering time was also increased by 2.9% on average and 8% on maximum, illustrating the overhead of GPU virtualization with containers.

It is also worth noting that container also improved performance in certain cases, as shown with the negative overheads in Figure 20. A preliminary analysis showed that these performance improvements were mainly due to containerization reduced resource contention among the benchmarks and VNC servers. Although further analysis is still required to identify the exact cause of the reduced contention, these performance improvements illustrate the potential benefits of container-based run-time optimizations.

Container Overhead Summary 1) On average, Docker containers incur limited overhead, further showing the feasibility of executing 3D applications in the cloud. 2) Nonetheless, high performance overhead may still be observed in certain cases, suggesting that container overhead reduction is still required. 3) Container overheads are mainly associated with IPC and GPU virtualization. 4) Containerization may also improve performance, suggesting the potential of additional run-time optimizations.

VI. Optimized Frame Copy

As discussed in Section V-A2, the frame-copy (FC) stage was a major performance bottleneck in TurboVNC. This section presents the optimizations we invented and implemented to reduce the frame-copying time.

Further analysis of the TurboVNC's graphics interposer revealed two inefficiencies. First, the interposer called the function *XGetWindowAttributes* before copying a frame. *XGetWindowAttributes* was extremely slow and consumed 6˜9ms. This function was only used to get the benchmark's resolution to determine the size of the frame to copy. As the resolution of a game or VR application is rarely changed during execution, there is no need to call *XGetWindowAttributes* for every frame copy. Therefore, in the first optimization, we applied memoization to this function. That is, we intercepted the invocation to *XGetWindowAttributes* and returned the cached resolution instead of actually calling it. *XGetWindowAttributes* is only actually invoked when the benchmark's resolution changes, which is determined by monitoring X events at Hook$_4$.

The second inefficiency is that the benchmarks were halted during the copy, waiting from the GPU to send the frame, as shown with the blank in the FC stage in Figure 5. Inspired by deep CPU pipelining, we broke the frame copy into two smaller steps – the start-copy and finish-copy. As shown in Figure 21, after issuing the frame copying command to GPU for frame$_{i-1}$ (FCStart$_{i-1}$), the graphics interposer does not wait for frame$_{i-1}$'s copy to finish. Instead, it goes on to finish the copying of frame$_{i-2}$ (FCEnd$_{i-2}$) and works on sending frame$_{i-2}$ to the VNC server. The actual finish of copying frame$_{i-1}$ happens after the application logic for frame$_{i+1}$ is computed (FCEnd$_{i-1}$). By making the frame copy into two asynchronous steps, the halt in the benchmark is removed.

Figure 22 gives the performance improvements from our two optimizations when one benchmark instance was executed. Our optimization improved server FPS by 57.7% on average and 115.2% at maximum. The client FPS was improved by 7.4% on average and 19.5% at maximum. The RTT was reduced by 8.5% on average and 15.1% at maximum. Note that, in Figure 22, the client FPS of *ITP* had 3% reduction due to the increased benchmark performance causing more contention with the VNC proxy. We were able to remove this extra contention and improve *ITP*'s client FPS with an additional optimization. However, due to space limitation, this additional optimization cannot be covered here.

VII. Related Work

VDI and Cloud Gaming System Benchmarking There have been several studies providing benchmarking tools or methodologies to analyze the performance of VDI systems and cloud gaming systems. Table IV compares the functionalities and features of the major prior work with Pictor. All of the studies summarized in Table IV, expect Chen et al. [15],

TABLE IV
COMPARISON BETWEEN PICTOR AND PRIOR WORK ON VDI AND CLOUD GAMING PERFORMANCE ANALYSIS.

Features	VNCPlay [85]	Chen et al. [15]	Slow-Motion [54]	Login-VSI [75]	DeskBench [66]	VDBench [12]	Dusi et al. [20]	Pictor
Random UI Objects Tolerant								✓
Varying Net Latency Tolerant	✓				✓	✓		✓
User-input Tracking			✓			✓		✓
CPU Perf. Measurement	✓	✓	✓	✓	✓	✓	✓	✓
Network Perf. Measurement	✓	✓	✓	✓	✓	✓	✓	✓
GPU Perf. Measurement								✓
PCIe frame-copy Perf. Measure.								✓
Unaltered 3D App Behaviors	✓			✓	✓		✓	✓

were designed to measure VDI systems with 2D applications. Moreover, none of these studies considered the random and irregular UI objects in cloud games and VR applications. And none of these studies provided methods to measure the performance of GPU rendering and frame-copy over the PCIe connections. These studies also did not provided means to track input processing without changing the 3D application's resource usage and behavior, as shown in Section IV. In summary, without the abilities to handle irregular and random objects, associate an input and its response, and measure the GPU and frame-copy performance, existing benchmarking tools/methodologies cannot provide reliable and effective performance measurements for cloud 3D rendering systems.

GPU Benchmarks. There are also many GPU benchmarks, such as GraalBench [10], SPECviewperf [18], GFXBench [33] Rodinia [14], and MGMark [76]. Mitra and Chiueh also analyzed three 3D benchmark suites [53]. These benchmarks and analysis focused on evaluating GPU performance without user actions. However, as 3D applications' behaviors are heavily affected by user actions, user inputs must be considered in cloud 3D benchmarks to ensure realistic results. Moreover, interactive 3D applications have intensive usage for both CPU and GPU. Only focusing on GPU cannot provide the insights needed to manage interactive 3D applications and their use of heterogeneous hardware.

Graphics Rendering Software. Many software supports remote desktops, such as VNC, NX and THINC [11], [64], [67], [69]. These remote desktops usually do not support 3D applications by default. Additionally, cloud graphics rendering also requires additional support on GPU virtualization and management for co-running 3D applications. Consequently, additional research is required to efficiently support 3D applications in cloud. CloudVR and Furion were two programming frameworks to support cloud VR [38], [42]. Abe et al. employed data prefetching to speed up the cloud processing time for interactive applications [3]. Ha et al. investigated the impact of consolidating multimedia and machine-learning applications in the cloud [26]. AppStreamer dynamically predicted and downloaded useful portions of a game to mobile devices [80]. EVR was a cloud VR system with specialized hardware to support 360° videos [43]. Meng et al. proposed to share the common backgrounds of multi-user VR applications to improve the performance for cloud VR [51]. Hegazy et al. proposed to allocate bit rates for encoded cloud gaming frames based on the importance of virtual objects [27]. There were also studies on the cloud server allocation and GPU virtualization for cloud gaming [19], [44], [45], [47], [86]. Our research is inspired by these studies and aims at facilitating these graphics system design studies. Google, Microsoft, NVidia, and PARSEC are also building their proprietary cloud gaming systems, whose designs may be different than the system analyzed in this paper [23], [52], [56], [61]. Pictor aims at facilitating the public research on cloud graphics rendering, so that open-source solutions can be as good as proprietary solutions. We will constantly update Pictor to follow the advances in these open-source solutions.

Other Related Work GUI testing frameworks [4], [5], [16], [46], [49] may also be used for benchmarking remotely-rendered applications. However, these GUI testing frameworks were not designed for 3D applications with irregularly-shaped and random UI objects. Google DeepMind and OpenAI Five have also built AI bots to play games [36], [59]. These bots were built to compete with human. Therefore, their execution may require thousands of processors [60]. Additionally, the AI models used by these bots required complex training processes for new games, and existing model are not publicly available. Therefore, these AI bots are not suitable for 3D application benchmarking. Moreover, prior research on non-cloud VR architecture and systems [9], [42], [43], [84] may also benefit from Pictor's intelligent clients and benchmarks.

VIII. CONCLUSION

This paper presents Pictor, a benchmarking framework for cloud 3D applications and systems. Pictor includes an intelligent client to mimic human interactions with 3D applications with 1.6% error, and a performance analysis framework that provides detailed performance measurements for cloud graphics rendering systems. With Pictor, we designed a benchmark suite with six 3D benchmarks. Using these benchmarks, we characterized a current cloud graphics rendering system and cloud 3D applications, which also showed benefits of cloud graphics rendering. We also designed new optimizations with Pictor to address two newly-found bottlenecks which improved the frame rate by 57.7% on average.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th*

USENIX Conference on Operating Systems Design and Implementation, 2016.

[2] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling High-Quality Untethered Virtual Reality. In *USENIX Symp. on Networked Systems Design and Implementation*, 2017.

[3] Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, H. Andrés Lagar-Cavilla, and Mahadev Satyanarayanan. vTube: Efficient Streaming of Virtual Appliances over Last-Mile Networks. In *Proc. of the Annual Symp. on Cloud Computing*, 2013.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software*, 32(5), 2015.

[5] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proc. of Int'l Conf. on Automated Software Engineering*, 2012.

[6] Amazon. Sumerian. https://aws.amazon.com/sumerian/. [Online; accessed 11-Nov-2018].

[7] AMD. GPU Performance API for AMD GPUs. https://github.com/GPUOpen-Tools/GPA. [Online; accessed 11-Jul-2019].

[8] AMD. ROCm System Management Interface (ROCm SMI) Library. https://github.com/RadeonOpenCompute/rocm_smi_lib. [Online; accessed 11-Jul-2019].

[9] M. Anglada, E. de Lucas, J. Parcerisa, J. L. Aragón, and A. González. Early Visibility Resolution for Removing Ineffectual Computations in the Graphics Pipeline. In *IEEE Int'l Symp. on High Performance Computer Architecture*, 2019.

[10] Iosif Antochi, Ben Juurlink, Stamatis Vassiliadis, and Petri Liuha. GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones. In *Proc. of the Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2004.

[11] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. Thinc: A virtual display architecture for thin-client computing. In *Proc. of ACM Symp. on Operating Systems Principles*, 2005.

[12] A. Berryman, P. Calyam, M. Honigford, and A. M. Lai. VDBench: A Benchmarking Toolkit for Thin-Client Based Virtual Desktop Environments. In *IEEE Int'l Conf. on Cloud Computing Technology and Science*, 2010.

[13] Y. S. Boger, R. A. Pavlik, and R. M. Taylor. OSVR: An Open-source Virtual Reality Platform for Both Industry and Academia. In *IEEE Virtual Reality (VR)*, 2015.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of Int'l Symp. on Workload Characterization*, 2009.

[15] K. Chen, Y. Chang, H. Hsu, D. Chen, C. Huang, and C. Hsu. On the Quality of Service of Cloud Gaming Systems. *IEEE Transactions on Multimedia*, 16(2), 2014.

[16] Wontae Choi, George Necula, and Koushik Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proc. of Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, 2013.

[17] Darrell Commander. VirtualGL: 3D without boundaries–the VirtualGL project, 2007.

[18] Standard Performance Evaluation Corporation. SPECviewperf 12. https://www.spec.org/gwpg/gpc.static/vp12info.html. [Online; accessed 11-Nov-2018].

[19] Yunhua Deng, Yusen Li, Xueyan Tang, and Wentong Cai. Server Allocation for Multiplayer Cloud Gaming. In *Proc. of ACM Int'l Conf. on Multimedia*, 2016.

[20] M. Dusi, S. Napolitano, S. Niccolini, and S. Longo. A Closer Look at Thin-client Connections: Statistical Application Identification for QoE Detection. *IEEE Communications Magazine*, 50(11), 2012.

[21] John Eidson and Kang Lee. IEEE 1588 Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *ISA/IEEE Sensors for Industry Conference*, 2002.

[22] Wildfire Games. 0 A.D. https://play0ad.com/. [Online; accessed 11-Nov-2018].

[23] Google. Stadia. https://stadia.dev/. [Online; accessed 10-Jul-2019].

[24] Khronos Group. Query Object. https://www.khronos.org/opengl/wiki/Query_Object. [Online; accessed 11-Jul-2019].

[25] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-Accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[26] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. The Impact of Mobile Multimedia Applications on Data Center Consolidation. In *IEEE Int'l Conf. on Cloud Engineering*, 2013.

[27] Mohamed Hegazy, Khaled Diab, Mehdi Saeedi, Boris Ivanovic, Ihab Amer, Yang Liu, Gabor Sines, and Mohamed Hefeeda. Content-aware video encoding for cloud gaming. In *Proc. of ACM Multimedia Systems Conference*, 2019.

[28] Joerg Henrichs. SuperTuxKart. https://supertuxkart.net/Main_Page. [Online; accessed 11-Nov-2018].

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural Computation*, 9(8), 1997.

[30] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

[31] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of ACM Multimedia Systems Conference*, 2013.

[32] The Khronos Group Inc. Vulkan 1.1.92 - A Specification. https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html. [Online; accessed 12-Nov-2018].

[33] Kishonti Informatics. GFXBench - Unified graphics benchmark based on DXBenchmark (DirectX) and GLBenchmark (OpenGL ES). https://gfxbench.com/. [Online; accessed 22-July-2019].

[34] Intel. Write Combining Memory Implementation Guidelines, 1998.

[35] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2016.

[36] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[37] Gareth R James. *Citrix XenDesktop Implementation: A Practical Guide for IT Professionals*. Elsevier, 2010.

[38] Teemu Kämäräinen, Matti Siekkinen, Jukka Eerikäinen, and Antti Ylä-Jääski. CloudVR: Cloud Accelerated Interactive Mobile Virtual Reality. In *Proceedings of the 26th ACM International Conference on Multimedia*, 2018.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

[40] H Andrés Lagar-Cavilla, Niraj Tolia, Mahadev Satyanarayanan, and Eyal De Lara. VMM-Independent Graphics Acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 33–43. ACM, 2007.

[41] Albert M. Lai and Jason Nieh. On the Performance of Wide-area Thin-client Computing. *ACM Transactions on Computer Systems*, 24(2), May 2006.

[42] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proc. of Int'l Conf. on Mobile Computing and Networking*, 2017.

[43] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. Energy-Efficient Video Processing for Virtual Reality. In *Proc. of Int'l Symp. on Computer Architecture*, 2019.

[44] Yusen Li, Yunhua Deng, Xueyan Tang, Wentong Cai, Xiaoguang Liu, and Gang Wang. Cost-efficient server provisioning for cloud gaming. *ACM Trans. Multimedia Comput. Commun. Appl.*, 14(3s), June 2018.

[45] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. GAugur: Quantifying Performance Interference of Colocated Games for Improving Resource Utilization in Cloud Gaming. In *Proc. of Int'l Symp. on High-Performance Parallel and Distributed Computing*, 2019.

[46] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting Concurrency for Android Applications Through Refactoring. In *Proc. of Int'l Symp. on Foundations of Software Engineering*, 2014.

[47] Q. Lu, J. Yao, H. Guan, and P. Gao. gQoS: A QoS-Oriented GPU Virtualization with Adaptive Capacity Sharing. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):843–855, 2020.

[48] D. Luebke and G. Humphreys. How GPUs Work. *IEEE Computer*, 40(2), Feb 2007.

[49] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proc. of Int'l Symp. on Software Testing and Analysis*, 2016.

[50] Joshua McCoy and Michael Mateas. An integrated agent for playing real-time strategy games. In *AAAI*, volume 8, pages 1313–1318, 2008.

[51] Jiayi Meng, Sibendu Paul, and Y. Charlie Hu. Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices. In *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2020.

[52] Microsoft. Project xCloud. https://www.xbox.com/en-US/xbox-game-streaming/project-xcloud. [Online; accessed 10-Jul-2019].

[53] Tulika Mitra and Tzi-cker Chiueh. Dynamic 3D Graphics Workload Characterization and the Architectural Implications. In *Proc. of the Int'l Symposium on Microarchitecture*, 1999.

[54] Jason Nieh, S. Jae Yang, and Naomi Novik. Measuring Thin-client Performance Using Slow-motion Benchmarking. *ACM Transactions on Computer Systems*, 21(1), February 2003.

[55] Nival. InMind VR. https://luden.io/inmind/. [Online; accessed 22-July-2018].

[56] NVIDIA. Geforce Now. https://www.nvidia.com/en-us/geforce/products/geforce-now/. [Online; accessed 11-Nov-2018].

[57] NVidia. NVIDIA Container Toolkit. https://github.com/NVIDIA/nvidia-docker. [Online; accessed 08-Aug-2019].

[58] NVIDIA. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml. [Online; accessed 11-Jul-2019].

[59] OpenAI. OpenAI Five. https://openai.com/projects/five/. [Online; accessed 11-Apr-2020].

[60] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[61] PARSEC. Game, Work, and Play Together from Anywhere. https://parsecgaming.com/. [Online; accessed 11-Apr-2020].

[62] Brian Paul. The Mesa 3D Graphics Library. https://mesa3d.org/. [Online; accessed 12-Nov-2018].

[63] Micha Pfeiffer, Hannes Kenngott, Anas Preukschas, Matthias Huber, Lisa Bettscheider, Beat Müller-Stich, and Stefanie Speidel. IMHOTEP: virtual reality framework for surgical applications. *International Journal of Computer Assisted Radiology and Surgery*, 13(5), May 2018.

[64] Gian Filippo Pinzari. Introduction to NX technology, 2003.

[65] Quinton Reeves and Lee Salzman. Red Eclipse: A Free Arena Shooter Featuring Parktour. https://www.redeclipse.net/. [Online; accessed 11-Nov-2018].

[66] Junghwan Rhee, A. Kochut, and K. Beaty. DeskBench: Flexible Virtual Desktop Benchmarking Toolkit. In *International Symposium on Integrated Network Management*, 2009.

[67] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), Jan 1998.

[68] Francis Rumsey, Peter Ward, and Slawomir K Zielinski. Can Playing A Computer Game Affect Perception of Audio-Visual Synchrony? In *Audio Engineering Society Convention*. Audio Engineering Society, 2004.

[76] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Rafael Ubal, Xiang Gong, Shane Treadway, Yuhui Bao, Vincent Zhao, José L. Abellán, John Kim, Ajay Joshi, and David R. Kaeli. MGSim + MGMark:

[69] Robert W Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2), 1986.

[70] Henning Schulzrinne, Anup Rao, and Robert Lanphier. Real Time Streaming Protocol (RTSP). Technical report, 1998.

[71] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.

[72] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proc. of Int'l Conf. on Machine Learning*, 2011.

[73] Valve Software. OpenVR SDK. https://github.com/ValveSoftware/openvr. [Online; accessed 12-Nov-2018].

[74] A. Sperduti and A. Starita. Supervised Neural Networks for the Classification of Structures. *IEEE Transactions on Neural Networks*, 8(3), 1997.

[75] R Spruijt, J Kamp, and S Huisman. Login Virtual Session Indexer (VSI) Benchmarking. *Virtual Reality Check Project-Phase II Whitepaper*, 2010. A Framework for Multi-GPU System Research. *CoRR*, abs/1811.02884, 2018.

[77] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting Mobile VR in LTE Networks: How Close Are We? *Proc. ACM Meas. Anal. Comput. Syst.*, 2(1), April 2018.

[78] Unity Technologies. Unity. https://unity3d.com/. [Online; accessed 12-Nov-2018].

[79] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[80] Nawanol Theera-Ampornpunt, Shikhar Suryavansh, Sameer Manchanda, Rajesh Panta, Kaustubh Joshi, Mostafa Ammar, Mung Chiang, and Saurabh Bagchi. Appstreamer: Reducing storage requirements of mobile games through predictive streaming. In *Proc. of Int'l Conf. on Embedded Wireless Systems and Networks*, 2020.

[81] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems 30*. 2017.

[82] TurboVNC. TurboVNC. https://turbovnc.org/. [Online; accessed 22-July-2018].

[83] Valve. InMind VR. http://blog.dota2.com/?l=english. [Online; accessed 22-July-2018].

[84] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song. PIM-VR: Erasing Motion Anomalies In Highly-Interactive Virtual Reality World with Customized Memory Cube. In *IEEE Int'l Symp. on High Performance Computer Architecture*, 2019.

[85] Nickolai Zeldovich and Ramesh Chandra. Interactive Performance Measurement with VNCplay. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[86] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan. vGASA: Adaptive Scheduling Algorithm of Virtualized GPU Resource in Cloud Gaming. *IEEE Transactions on Parallel and Distributed Systems*, 25(11), 2014.