

# OpenUVR: an Open-Source System Framework for Untethered Virtual Reality Applications

Alec Rohloff<sup>§</sup>  
Applied Research Associates

Zackary Allen<sup>§</sup>  
Red Hat, Inc.

Kung-Min Lin<sup>§</sup>  
University of California, Berkeley

Joshua Okrend<sup>§</sup>  
Riverside Technology, Inc.

Chengyi Nie<sup>§</sup>  
Stony Brook University, New York

Yu-Chia Liu  
University of California, Riverside

Hung-Wei Tseng  
University of California, Riverside

**Abstract**—Advancements in heterogeneous computing technologies enable the significant potential of virtual reality (VR) applications. To offer the best user experience (UX), a system should adopt an untethered, wireless-network-based architecture to transfer VR content between the user and the content generator. However, modern wireless network technologies make implementing such an architecture challenging, as VR applications require superior video quality—with high resolution, high frame rates, and very low latency.

This paper presents OpenUVR, an open-source framework that uses commodity hardware components to satisfy the demands of interactive, real-time VR applications. OpenUVR significantly improves UX through a redesign of the system stack and addresses the most time-sensitive issues associated with redundant memory copying in modern computing systems. OpenUVR presents a cross-layered VR datapath to avoid redundant data operations and computation among system components, OpenUVR customizes the network stack to eliminate unnecessary memory operations incurred by mismatching data formats in each layer, and OpenUVR uses feedback from mobile devices to remove memory buffers.

Together, these modifications allow OpenUVR to reduce VR application delays to 14.32 ms, meeting the 20 ms minimum latency in avoiding motion sickness. As an open-source system that is fully compatible with commodity hardware, OpenUVR offers the research community an opportunity to develop, investigate, and optimize applications for untethered, high-performance VR architectures.

## I. INTRODUCTION

With high-performance graphics processing units (GPUs) and hardware accelerators, modern heterogeneous personal computers already have the computing power needed for virtual reality (VR) applications. But latency and mobility are still a problem; to provide the best user experience (UX), a system must be enabling the user to move around freely and able to deliver VR content to the user in 20 ms to avoid motion sickness in real-time gaming [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Unfortunately, existing commercialized VR solutions adopt either a tethered architecture that limits user

mobility [11], [12] or an untethered architecture (using mobile devices) that limits visual effects [13], [14], [15], [16].

The major obstacle in developing effective untethered VR (UVR) systems lies with a mismatch between the bandwidth demand for high-quality visual content and the sustainable bandwidth provided by the underlying wireless VR links. Though compression helps lower the bandwidth demand to tens of Mbps, the increased visual latency is unacceptable to latency-sensitive VR applications. Recent research projects have promoted millimeter-wave (mmWave) wireless technologies to avoid the trade-off between latency and bandwidth [17], [18], [19]. However, such technologies severely limit user mobility because they depend on line-of-sight signaling. As a result, currently available UVR products can offer only mildly entertaining novelty titles.

This paper presents OpenUVR, a fully open-source, open-architecture VR system framework that resolves the three-way trade-off between latency, bandwidth, and UX. Rather than treating a VR application as a type of video streaming with bandwidth issues, as is often done, OpenUVR improves the end-to-end latency that is critical to real-time, interactive action titles. OpenUVR also builds on an important lesson from modern high-performance I/O systems—that redundant memory copies comprise the most significant part of data-exchange latency [20], [21]. We have found memory copies from three sources to be especially problematic for UVR architectures: (1) data exchanges between memory spaces, (2) data transformations between system modules/stacks, and (3) data buffering between system components.

OpenUVR's design revisits the complete datapath (including graphics/video libraries and the network stack), and accounts for application demands in order to minimize the number of memory copies in delivering VR content. The OpenUVR API and kernel modules work with commodity hardware components to eliminate redundant memory buffers/operations and avoid context switches before data is sent to or received from network stacks. OpenUVR further simplifies the network stack by using peer-to-peer network architecture and abandoning the traditional multilayer network-stack delivery design that regenerates packets to accommodate the headers of each layer.

Eliminating unnecessary memory copies allows OpenUVR to achieve an imperceptible delay of just 14.32 ms in real-time VR applications—and this is done with affordable components and current-generation Wi-Fi technologies. And because the OpenUVR mobile user device (MUD) performs minimal

The authors would like to thank the anonymous reviewers for their helpful comments. We also owe a debt of gratitude to Christopher Fraser for his excellent copyediting skills. This work was partially sponsored by the two National Science Foundation (NSF) awards, 1940048 and 2007124. This work was also supported by new faculty start-up funds from North Carolina State University and University of California, Riverside.

Zackary Allen and Joshua Okrend are former undergraduate researchers at North Carolina State University advised by Hung-Wei Tseng. Kung-Min Lin is a former summer intern at University of California, Riverside advised by Hung-Wei Tseng. Chengyi Nie is a former undergraduate researcher at University of California, Riverside advised by Hung-Wei Tseng.

computations, it can be powered by a simple Raspberry Pi single-board computer that costs only USD 35.

By exploring the unique properties, challenges, and trade-offs inherent in the design of a UVR system, this paper makes several contributions.

**(1) The paper reveals the inefficiencies of system stacks and the need for cross-layer optimizations in wireless VR systems.** As an experimental system, OpenUVR shows that the latency overhead from each layer in the system stack can be remedied through optimal buffering, memory copying, transcoding, and data exchange. Our observations suggest that an efficient VR system must present local optimizations in all layers instead of focusing on a single-point design.

**(2) The paper presents a general design philosophy for wireless networking applications that require low latency and high throughput.** OpenUVR's holistic, latency-mitigating system framework bypasses the inefficiencies of network stacks and uses existing hardware accelerators to yield a robust wireless VR infrastructure. The same efficient design can easily be applied to wireless systems with similar demands as an alternative to expensive, throughput-oriented wireless network technologies.

**(3) The paper advances research by providing a publicly available, open-source solution to VR system challenges.** OpenUVR's hardware components and wireless technologies are widely available and relatively low-cost, and its software may be freely inspected and modified. As a fully open-source solution, OpenUVR will help researchers extend and optimize VR architectures. §

## II. BACKGROUND

We now describe the basic architecture of a VR system and the challenges of building a VR system that promotes UX. We also discuss existing VR solutions and their associated drawbacks.

### A. Basic operations and challenges of VR systems

A typical VR system needs to accomplish three main types of tasks:

**User input collection** The system must gather all user inputs from the commands that users explicitly send through a device (e.g., a keyboard, microphone, or controller) and data from sensors that detect the user's motions.

**Content generation** The system must process user inputs, select the elements to appear in the VR content, and render the content according to application specifications. Rendering VR content requires high-performance computing resources and data retrieval from either local storage or a cloud service.

**Content presentation** The system must display the VR content to the user, usually as video frames and sound effects. The user then reacts to the VR content and generates corresponding inputs that initiate another task loop.

To deliver the best UX, a system needs to address three challenges when completing the aforementioned VR tasks:

**Visual effects** The system must render video frames that show views of virtual scenes. Such rendering

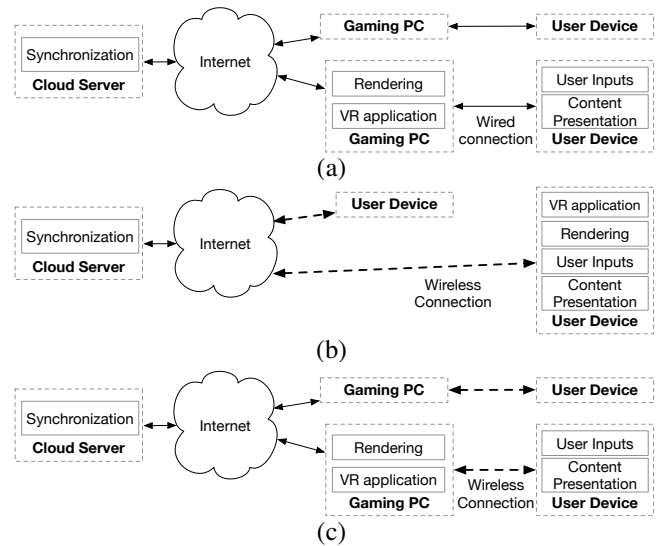


Fig. 1. The VR task allocations in different VR system architectures

requires high resolutions and high frame rates (on the order of 60 frames per second [FPS] in 1080p resolution for both eyes) [22].

**Response latency** A delayed response in immersive VR applications can cause motion sickness, so a VR system must complete all task loops with reasonably low latency (20 ms).

**Physical comfort** In many VR applications, especially games, the user can move about to perform VR actions, so the MUD, which sends/receives inputs and displays visual effects, must be lightweight and untethered to maximize comfort and minimize constraints.

Unfortunately, no existing commercial VR system rises to meet all of the above challenges.

### B. Existing VR systems

Since the release of Facebook's Oculus Rift VR headset in 2016 [11], VR systems have steadily gained market share as companies have introduced improved designs. Figure 1 depicts the three most common designs for modern VR systems and how these systems allocate VR tasks among their components.

**Tethered device** Figure 1(a) illustrates the system architecture of a VR system that uses a tethered device. A VR system relying on a tethered device consists of a high-performance PC and a low-performance wearable device. These two components are connected by wires that power the wearable device, transmit audio-visual output to the device, and carry user input back to the PC from the device. The Oculus Rift and the HTC Vive [12] are examples of such systems. The Rift only tracks the orientation of the headset, and the Vive uses room sensors to track the motion of the user in 3D space. In both systems, wires connected to the interactive device limit the user's movements, create a tripping hazard, and undermine the immersive experience at the heart of VR.

**Untethered device without a host PC** Figure 1(b) illustrates the concept behind this type of VR system. All-in-one devices place all VR tasks on the MUD, with the exception of social-synchronization tasks or leverage the cloud for rendering. MUDs fall into two main categories: MUDs

powered by smartphones and MUDs powered by standalone devices.

Smartphone-powered headsets such as the Google Daydream [13] and the Samsung Gear VR [14] rely on a smartphone's computing power to render images, the phone's screen to display the images, and the phone's accelerometers to track the user's motions. Since the Google and Samsung systems are limited by the capabilities of the user's smartphone, highly detailed VR games and experiences available on a PC are unavailable to users with low-end phones.

Standalone VR systems like the Oculus Go [15], the Oculus Quest, and the Lenovo Mirage Solo [16] are similar to smartphone-dependent VR devices, except that the Go and Solo have their own screens, sensors, and smartphone-grade processors to render and display images. As with the smartphone-dependent devices, the Go and Solo are unable to render highly detailed, complex VR experiences because the processors in their devices are so limited.

To enable complex VR experiences while maintaining the simplicity of MUDs, the recent cloud services including Google Stadia [23], Steam Remote Play [24], and NVIDIA's GeForce Now, adopts the system architecture of using the GPUs on cloud servers for rendering. To address the high latency due to the distance and unpredictable traffic condition between cloud servers and the MUD, the system intensively employs techniques including post-render warp [25], [26] that render only informative parts of the upcoming frames on the remote cloud. However, doing this requires significantly tuning on the game engine and the visual library, increasing the overhead of game development while the MUD still suffers from the long latency with cloud servers.

**Untethered device with a host PC** The most promising approach to creating a fully immersive wireless experience on par with that of wired VR is to render visuals on a separate, powerful computer and stream the visuals to a wearable device using a fast wireless network (e.g., a network based on IEEE 802.11ac or 60 GHz mmWave solutions).

A few startup companies like Amimon, KwikVR, and TPCAST (and some research groups as well) are developing such UVR systems using 60 GHz mmWave wireless networks; however, the systems haven't yet reached the wider market due to the limitations of mmWave wireless technologies. TPCAST advertises a latency of less than 10 ms using a 60 GHz wireless connection, but a 60 GHz signal suffers from very high attenuation. Furthermore, a 60 GHz signal may rely on beam focusing to overcome attenuation, so users can easily lose the wireless connection for a VR application when they move the head-mounted display (HMD) outside of the transmission beam. MoVR [19] tries to address the limitation of 60 GHz mmWave with a specially designed antenna and mirrors carefully positioned in the VR environment; this is far from ideal, as MoVR still requires special environment to work. Zhong et. al. [27] and Liu et. al. [18] investigate the use

of 60 GHz wireless networks and rendering pipelines placed between the host PC and the receiver, but 60 GHz wireless signals have a high decoding rate and have a severely limited range. These UVR systems also rely on laptop PCs that fail to match the power, weight, and budget limitations of a MUD.

As with OpenUVR, Furion [2] leverages mature commodity IEEE 802.11ac technology to transmit content from the host PC. By parallelizing the video codec and splitting background/foreground rendering, Furion minimizes latency to the same level as OpenUVR. However, Furion's approach requires multiple fully active processor cores on both the host PC and MUD. As a result, a UVR system with Furion still needs a smartphone as a MUD. GamingAnywhere [28] is an earlier open-source game-streaming system design that can be used directly with an IEEE 802.11ac wireless network. Like Furion, GamingAnywhere requires a MUD with significant computing power. Nonetheless, GamingAnywhere is an open-source project, and the system's inefficiencies have benefitted OpenUVR's design.

### III. PERFORMANCE OF MODERN UVR SYSTEMS

In this section, we evaluated the performance, particularly the visual latency between the host PC and the MUD, of system frameworks that can carry VR content untetheredly without relying on proprietary hardware components. Our evaluation shows that all existing system frameworks fail to meet the desired 20 ms deadline in avoiding motion sickness. We further investigated the latency in each component of the fully open-sourced GamingAnywhere platform [28] and figured out the performance bottleneck in existing systems.

#### A. Experimental methodology

This section describes the hardware platforms that we used to build the host PC and the MUD for selected frameworks as well as the proposed OpenUVR that this paper will introduce later. We also explain how we measured latency in all evaluated systems.

1) *Evaluated system platforms:* We evaluated three platforms that can generate VR content on a desktop gaming PC and communicate with an MUD through wireless links. We selected these platforms as they (1) rely on modern, commercialized wireless technologies, (2) all support Unreal engine to enable fair comparisons, and (3) at least allow the server software installed on a custom-built gaming desktop.

These three platforms are (1) NVIDIA's Moonlight [29] that can use either a laptop or a RPi 4b as MUD with only the client software open-sourced. (2) SteamLink [30] that uses a dedicated wireless end-user device with an open-accessed API. (3) GamingAnywhere [28] that can use either a laptop or RPi 4b as MUD, but fully open-sourced on both desktop PC server and MUD clients. In addition to the above three, we also evaluated (4) HWA-GamingAnywhere, an improved version of GamingAnywhere that we modified to leverage NVIDIA's hardware accelerated NVENC on the host PC/laptop. On the RPi4b MUD, HWA-GamingAnywhere uses the OpenMAX Bellagio IL on [31] to write an H.264 packet into GPU memory and use the on-chip VideoCore IV accelerator to decode and directly render frames to the screen.

2) *Experimental platform/target hardware architecture:* In all evaluated UVR architectures, the host PC uses a middle-class CPU and a high-performance GPU, an assembly similar to most gaming desktop computers, to generate VR content. The MUD of HWA-GamingAnywhere and OpenUVR, by contrast, consists of only a Raspberry Pi computer with a



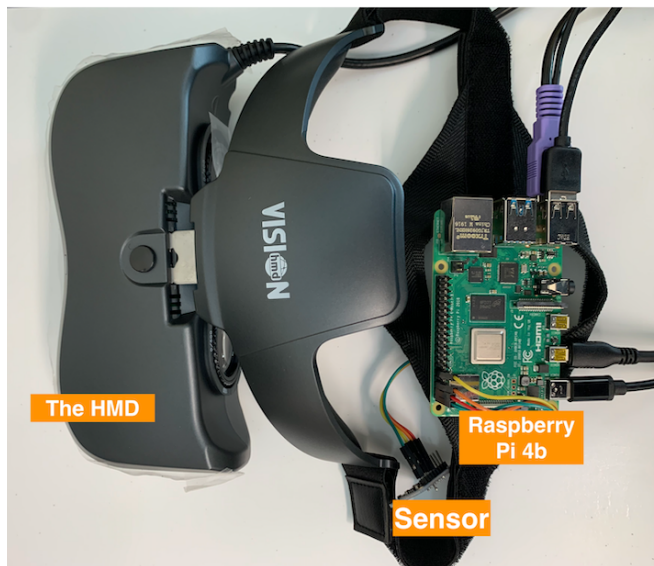


Fig. 2. The prototype MUD

wireless network-interface card (NIC). The wireless network uses IEEE 802.11ac technology. When we tested each UVR framework, we found that some interference from other wireless networks was unavoidable; each use case operated within an environment that included 12 competing access points.

a) *The host PC*: We built a host PC with a quad-core Intel processor based on the Coffee Lake architecture. The processor runs at a maximum clock rate of 4.6 GHz and dynamically adjusts the voltage between 0.8 GHz and 4.6 GHz. The system has 16 GB DDR4 DRAM for main memory and an NVIDIA RTX 2080 GPU attached to the host machine via 16× PCIe Gen.3 lanes. The NVIDIA RTX 2080 GPU provides hardware-accelerated NVENC encoder that evaluated UVR frameworks can leverage. To provide wireless network capabilities, we installed a PCIe wireless card with a Qualcomm Atheros chip and a 2×5dBi omni-directional, external, detachable, dual-band antenna supporting the IEEE 802.11ac standard at 867 Mbps. The host PC applies the default power saving mode in the installed Ubuntu system.

b) *The MUD*: The target MUD architecture in OpenUVR uses a Raspberry Pi 4 Model B (RPI4b) single-board computer that costs only USD 35 [32]. RPI4b runs a simplified Linux distribution (Raspbian) on a Broadcom BCM2711. The SoC runs at 1.5 GHz with built-in GPU cores and has hardware accelerators for decoding H.264 video/audio. The RPI4b wireless NIC supports the IEEE 802.11ac standard. The MUD can output VR content to a wearable display like the VISIONHMD Bigeyes H1 584PPI 2.5K Screen 3D Video Glasses we used to test HWA-GamingAnywhere and OpenUVR. The MUD uses Raspbian's default power saving mode. We also installed gyroscopes and accelerometers on the MUD and wrote our own programs to translate sensed signals into meaningful movements and operations inside games. Figure 2 shows the prototype.

Although this work targets at system frameworks using lightweight MUD based on small and simple devices like RPI 4bs, we also tested the performance on a Dell Inspiron

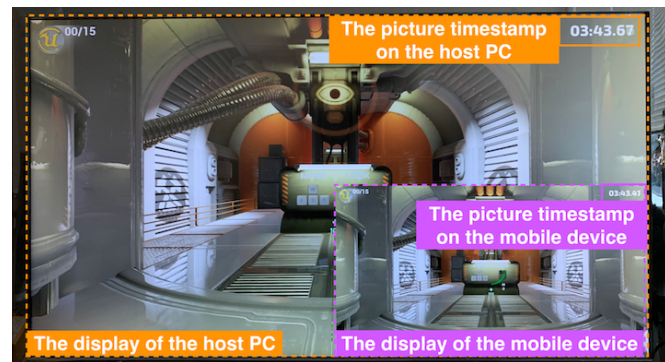


Fig. 3. A side-by-side comparison of terminal output from the host PC (orange) and the MUD (pink)

15 7559 laptop that uses an Intel Quad Core i7-6700HQ processor with a dedicated NVIDIA Geforce GTX 960M GPU as a reference design on MUD with intensive computing power.

3) *Performance measurements*: In evaluating the performance of VR systems, the most important metric is the end-to-end latency between VR-content generation and VR-content presentation. Unfortunately, fine-grain synchronization (e.g., in  $\mu$ s) of clocks on different machines is impossible, so simply comparing the received packet timestamp with the time on a different machine is not a feasible solution.

We addressed the difficulty of measuring the latency between the host PC and the MUD (or the set top box) by using photography (Figure 3). We first turned on the timestamp feature in each game so that each rendered image would contain a timestamp when generated on the host PC. We then displayed each image on a monitor connected to the host PC and a separate monitor connected to the MUD, with the displays from the two devices presented side-by-side. By using high-speed photography to compare timestamps shown on each display, we were able to calculate the latency between different nodes. As shown in Figure 3, the host PC's timestamp is 03:43.67, but the timestamp from the MUD is 03:43.41, reflecting a latency of 260 ms. These photos also helped us assess the quality of delivered picture frames; for each framework being tested, we randomly sampled 1,000 frames.

By default, we tested the performance of these frameworks using Unreal Tournament [33], an open-source, first-person shooter game. Though Unreal Tournament is not, strictly speaking, designed for VR, the game is by far the best choice for testing a VR system for the following reasons: (1) Unreal Tournament's first-person shooter gameplay allows the user to easily perceive visual delays. (2) Unreal Tournament runs on an open-source game engine (Unreal Engine), which makes optimization and code instrumentation easy. (3) Many popular VR systems like the Oculus and HTC Vive use Unreal Engine.

To guarantee that motion was encoded when performance was measured, we used scripts that emulate player input by causing a character to spin counterclockwise at a constant rate while standing in place and keep moving around in the scene for 10 minutes. This virtual action effectively simulates the user behavior commonly found in VR applications.

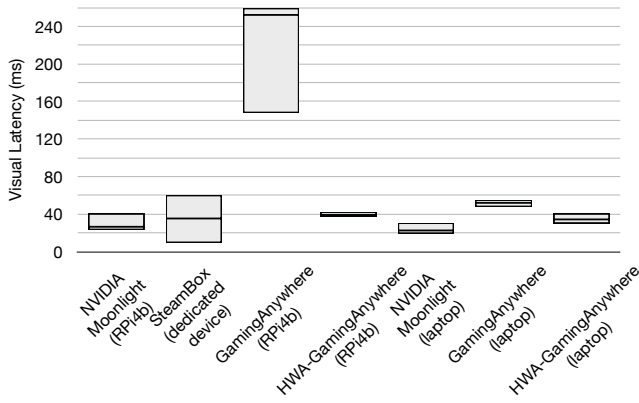


Fig. 4. Visual latency of evaluated platforms

### B. The Performance of Existing Systems

Figure 4 shows the visual latency of evaluated platforms. In general, platforms using laptops performs significantly better visual latency than using RPi 4bs due to the more powerful computing power on laptops. The original GamingAnywhere uses software video codec and the latency is  $1.96\times$  longer than Moonlight (i.e., 52 ms v.s. 26.5 ms) with the same client laptop. With the help of NVENC, HWA-GamingAnywhere delivers almost similar visual latency as Moonlight, at 28 ms on average, when using a laptop as the MUD. Steamlink's dedicated device sometimes outperforms the visual latency of using laptops, but still longer than Moonlight with a laptop MUD at 35 ms on average. When Moonlight and HWA-GamingAnywhere use RPi 4bs as MUDs, the visual latency of both platforms increases compared with using laptops. Moonlight achieves an average of 36 ms and HWA-GamingAnywhere achieves around 39.41 ms. Without NVENC's hardware acceleration, GamingAnywhere's latency on RPi 4b is as high as 1.2 sec. However, no matter using what type of MUD, all platforms' average visual latency failed the recommended 20 ms threshold to prevent motion sickness in real-time gaming. As a reference, the average latency of tethered systems like Oculus Rift [34] is around 4 ms.

### C. The Sources of Latency

To figure out the sources of failed end-to-end latency for real-time VR applications, we instrumented code of HWA-GamingAnywhere system in (a) the host game application, (b) the game engine and the underlying library functions and system stack modules, and (c) the software stack on the MUD. We only investigated HWA-GamingAnywhere in deep and believe that a detailed analysis of the latency in HWA-GamingAnywhere is valuable for designing UVR systems that lack laptop-class computing power on the mobile user side as (1) none of the platforms have reference designs that are fully open sourced, (2) HWA-GamingAnywhere delivers better performance as other counterparts, and (3) we instrumented the Moonlight's MUD code and the system implementation resembles HWA-GamingAnywhere, showing HWA-GamingAnywhere's system architecture is representative.

1) *Datapath of a frame in HWA-GamingAnywhere*: Figure 5 illustrates the datapath of a picture frame and the frame's audio streaming from the host PC to the mobile device. UVR Frameworks with a host PC that Section II-B describes [2], [19], [27], [18], [28] all use the same datapath. (Note that circled numbers in subsequent paragraphs correspond to the steps in Figure 5.)

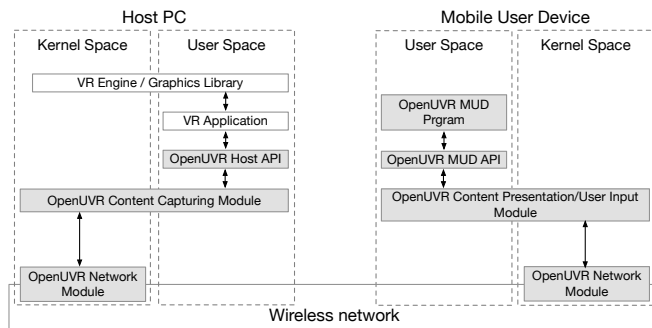
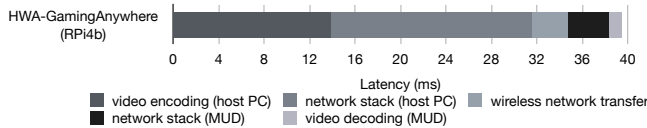
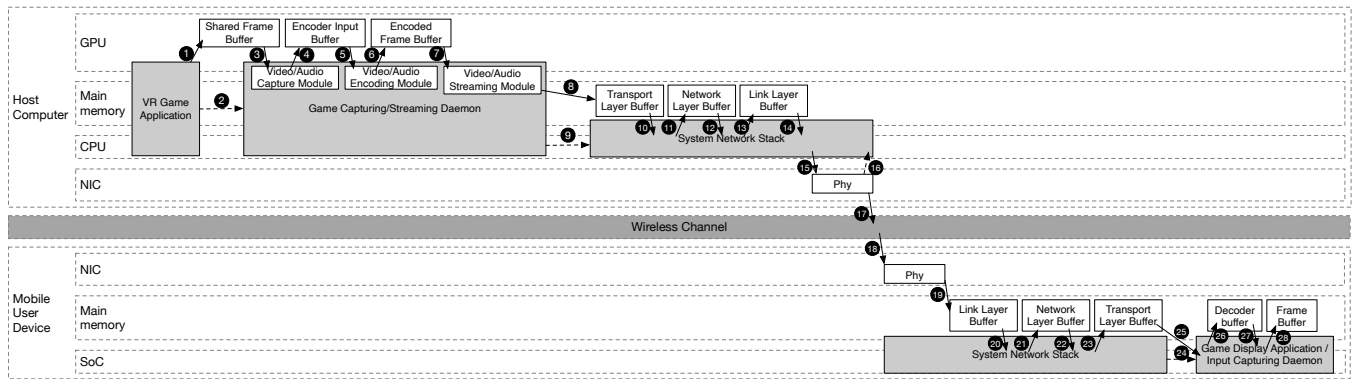
When a VR application generates a picture frame ①, the application signals ② a gaming capturing/streaming daemon (usually a separate process that is allowed to access a shared frame buffer within the GPU) to retrieve the generated content ③. To reduce the volume of encoded content, UVR frameworks convert the captured content from RGB format into YUV color space and place the converted content into an input buffer ④; this is typically done in GPU device memory to exploit the on-GPU hardware codec (e.g., NVIDIA's GPU-accelerated NVENC video/audio encoder).

Next, the UVR framework invokes the video/audio codec library to encode the transcoded content ⑤ and places the result into another buffer ⑥. Though modern GPUs support 4K resolutions, we chose the default 1080p resolution that modern VR systems use, as prior work suggests that 1080p resolution is sufficient for HMD [22]. Further, our empirical study and work by Bao et al. [1] show that a 20 Mbps bitrate effectively balances video/audio quality with bandwidth demand.

To deliver the encoded content to the destination mobile device, the UVR framework uses a streaming library to feed content into the network stack's transport layer (⑦ and ⑧) and signals the network stack to take over ⑨. Data must then pass through the network layer (⑩ and ⑪) and the link layer (⑫ and ⑬) before reaching the network-interface hardware (steps ⑭ through ⑰). Each network layer attaches its own protocol-specific header and checksum and repartitions the data into the layer's own protocol-specific format and packet size. For example, UDP specifies a 65,507-byte packet size, IPv4 specifies a 2,480-byte packet size, and IEEE 802.11ac specifies a 2,304-byte packet size. Consequently, CPU processing and multiple main-memory buffers are needed between steps ⑧ and ⑮.

Once the network-interface hardware on the mobile device has captured the transmitted content ⑱, data move back up the network stack to the transport layer (⑲ through ⑳). Content then moves from the transport-layer buffer to the user-end application, and the user-end daemon receives a signal to take over processing (⑲—㉔ and ㉕). The user-end daemon uses its own computing resource (typically a system-on-chip [SoC] with combined CPU and GPU cores) to decode the content (㉖ and ㉗). The daemon then hands off the decoded framework and audio ㉘ to an HMD-accessible buffer ㉙.

To improve the throughput and utilization of system components, modern UVR systems like our own exploit pipeline parallelism (Figure 5). But pipelining does not eliminate the need for each frame to go through every step in the datapath outlined above. Rather, pipelining increases latency due to the overhead of synchronizing pipeline stages.



2) *Latency analysis of HWA-GamingAnywhere*: Figure 6 breaks down the latency in each stage of the VR system pipeline. The most significant source of latency comes from the network stack on the host PC, which includes the transport-layer protocol (i.e., RTP/RTSP), the network-layer protocol (i.e., IPv4), and the link-layer soft MAC firmware. The network stack on the host PC contributes 17.63 ms to the end-to-end latency on our testbed. The video encoding latency, which contributes 13.94 ms to the total latency, ranks in second place. The aggregate latency measured in the host PC software stack is 31.57 ms.

On the other hand, the fixed cost in the network subsystem is only 3.2 ms; this includes propagation delay, physical modulation/demodulation, and link-layer control. The total latency on the MUD is only 3.64 ms, as HWA-GamingAnywhere requires minimal computation from the MUD while utilizing a highly optimized SoC.

pdfElement  
試用版

#### IV. OPENUVR SYSTEM DESIGN

With lessons learned from implementing HWA-Gaming-Anywhere, we identified the potential of fulfilling the 20 ms threshold that prevents motion sickness through simply reducing datapath software latency. In response to the potential points of optimizations, this paper presents the OpenUVR system. Figure 7 shows the system stack of OpenUVR. On the host-PC side, OpenUVR receives data from an application through the OpenUVR API, uses the OpenUVR content-capturing module to encode VR content, and delivers the encoded content to the MUD via the OpenUVR network module. Once the network module has received data on the MUD side, the OpenUVR content-presentation module will decode the content and allows the user-space MUD program to display the decoded data to the user. If the user experiences any feedback through sensors or a joystick, the MUD program captures the input and sends it to the host PC in reverse order.

In designing each OpenUVR module, we carefully optimized the software latency to eliminate (1) unnecessary data transcoding, (2) unnecessary memory copies, and (3) system-level control overhead. The system-stack architecture allows each framework to flow through the datapath shown in Figure 8—a datapath that is significantly more streamlined than the datapath in Figure 5. The following sections describe our design and the techniques we used to reduce latency.

### A. OpenUVR content-capturing module

Because the OpenUVR content-capturing module runs on the host PC, the module can intercept VR-application content while working closely with real-time video/audio-compression hardware accelerators before delivering the encoded content to the network module. The OpenUVR content-capturing module includes three sources of optimization to minimize unnecessary operations and reduce latency: transcoding avoidance, plug-in modules, and kernel-space functions.

1) *Transcoding avoidance*: In HWA-GamingAnywhere, the largest contributor to latency in the game-streaming daemon arises from converting the GPU-generated VR content from RGB format to YUV format. (YUV format is generally preferred for video-compression applications because it allows efficient lossy compression to reduce video size without having an appreciable effect on perceived video quality.) HWA-GamingAnywhere, Furion and GamingAnywhere use YUV420, which splits an image into its components of luma



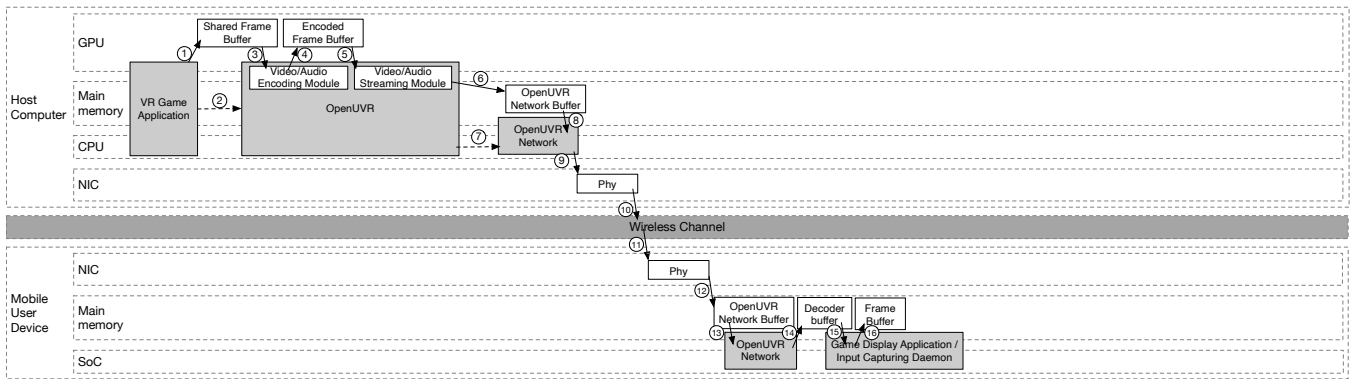


Fig. 8. The OpenUVR datapath

(black and white intensity) and 2-dimensional chrominance (color). YUV420 downsamples both chrominance components in the horizontal and vertical dimensions, thereby reducing the data required for each component by 50%.

In spite of the advantages that YUV420 offers, conversion between two different color spaces can incur considerable overhead. YUV420 may therefore contribute to latency between content generation and content perception. Additionally, color-space conversion not only consumes computing resources and time, but also requires memory space and memory access to accommodate the raw RGB content and the converted YUV content (steps ④ and ⑤ in Figure 5). In conventional video-streaming applications, the added latency from color-space conversion and buffering is acceptable because these applications can tolerate longer delays or do not require a user to interact with the content generator. Such applications compensate for the delays from color-space conversions by relying on higher video quality and lower bandwidth usage as well.

As VR applications are extremely latency sensitive, OpenUVR avoids color-space conversions, and so also avoids color-space transcoding. OpenUVR can leverage the hardware-accelerated encoder in the graphics card to compress VR content while the content is in native RGB format. OpenUVR then relies on the hardware accelerator in the RPi4b MUD's SoC to decode the RGB data. In this way, OpenUVR is able to simplify the process of generating encoded content from steps ③ through ⑥ in Figure 5 to just two steps—steps ③ and ④ in Figure 8.

2) *Plug-in modules for VR applications:* Unlike conventional VR solutions that use a daemon to capture application content, OpenUVR requires an application to either directly invoke OpenUVR's API functions or indirectly invoke them through a plug-in module in the game engine. The OpenUVR API functions do not create additional processes in the system. Rather, the functions all work within the same process, so they share the address space of the calling application. In this way, OpenUVR avoids the context-switching overhead and memory copies associated with inter-process communication. OpenUVR's signaling between the VR application and OpenUVR modules is more efficient.

Because OpenUVR and the VR game share both the host OS and GPU virtual memory space, the device memory locations

on hardware accelerators are visible to the OpenUVR content-capturing module. OpenUVR takes advantage of this visibility and avoids redundant memory copies by integrating hardware-accelerated codecs. Since a VR application can associate a GPU-device memory buffer with an OpenUVR context (which allows the application to share the same memory space as the context), OpenUVR can poll the GPU memory buffer periodically and invoke the video-encoding library on the GPU memory buffer directly; OpenUVR simply needs to move the final encoded VR content into the main memory minus the raw VR content that the GPU generates. Without this optimization, the current GPU runtime would need to either explicitly copy raw rendered data from the GPU memory to a shared memory location (e.g., texture memory) or use inter-process communication facilities (most likely the host DRAM).

3) *Kernel-space functions:* The OpenUVR content-capturing/presentation modules implement some of their functions as system kernel modules. For instance, after the API has initialized the OpenUVR context, the content-capturing module acts as a kernel module when it remaps the user-space main-memory locations to the same physical memory locations as those accessed by the system wireless-network stack (e.g., the NIC driver). This design allows the application data to directly reach the network stack without additional copies from user-space to kernel-space (steps ⑥ and ⑭ in Figure 8).

#### B. Wireless network subsystem

To tackle the most significant latency in the VR datapath, the wireless network subsystem in OpenUVR applies several design decisions that directly or indirectly lead to latency reduction.

1) *Direct peer-to-peer network topology:* OpenUVR adopts the peer-to-peer (P2P) network topology specified in IEEE 802.11ac—a topology that allows the host computer and MUD to directly communicate with each other (unlike the conventional infrastructure mode that relies on a base station/router as an intermediary for wireless communication).

Using P2P mode provides three benefits. First, the mode reduces the latency on wireless channels by eliminating the one-hop delay on the wireless base station. Though this latency is only 3.2 ms in HWA-GamingAnywhere, the latency can double as OpenUVR generates RGB encoded data that can

increase the data size of each frame. Second, P2P mode halves congestion on wireless channels because the host PC and the MUD usually share the same access point in HWA-GamingAnywhere and OpenUVR; using P2P mode eliminates the need for a datagram to go through the wireless channels within the same area twice. Third, direct communication between node pairs using dedicated channels removes the demands of the network-layer protocol; this, in turn, gets rid of memory copies from packet reformatting (steps 10 and 11 in Figure 5). In the current OpenUVR implementation, peers are connected on an 80-MHz-wide channel within the 5 GHz spectrum, and hostapd is configured to select the channel on which it detects the least interference from other radio broadcasts.

2) *Direct network I/O—OpenUVRDPP*: The OpenUVR data plane protocol (OpenUVRDPP) is customized to eliminate memory copies when data packets are reformed and headers attached in each layer of data-plane traffic. As real-time gaming is especially latency sensitive, the best transport-layer protocol for data traffic should neither perform additional operations nor use additional buffers (essentially a UDP protocol with no-ops). P2P network topology removes the need for the network-layer protocol and for packet reformation, allowing OpenUVR to bypass these two layers and minimize latency.

So instead of using the port number abstraction of UDP/RTP, OpenUVRDPP works with the OpenUVR kernel module to map the physical memory location of the data-link-layer buffer to the VR application's memory space. Given such access, OpenUVR can copy GPU data directly into OpenUVRDPP's data-link-layer buffer (step 8 in Figure 8) and generate packets without going through any conventional routing- and transport-layer protocols.

3) *Connection/quality control—OpenUVRCP*: To set up an OpenUVR connection and provide basic quality control, OpenUVR uses an off-band control protocol (OpenUVRCP). OpenUVRCP is relatively less time critical and less bandwidth-consuming than OpenUVRDPP. Thus, we did not implement a transport-layer bypass. Instead, we adopted UDP to carry OpenUVRCP control messages.

a) *Connection setup*: When a MUD launches its instances and grants access to the host PC's P2P network, the MUD sends an OpenUVRCP message to the host PC to request access. If the user chooses to grant the MUD's access request, the user adds the MUD's MAC address to the whitelist or goes through the host PC's administrative user interface.

b) *Quality control*: As OpenUVR still relies on video compression to reduce bandwidth demand, video-codec parameters play an important role in the trade-offs between latency, visual quality, and network bandwidth. Among the most important parameters is the size of a group of pictures (GOP). In the H.264 format that OpenUVR currently adopts, a GOP consists of an I-frame followed by a certain number

size defines the number of P-frames between each I-frame. P-frames are smaller than I-frames (about a quarter the size of an I-frame), so at a given constant target bitrate, having a larger GOP size should result in smaller data size; however, if the client drops a frame, the display will be corrupted for all following P-frames because each frame builds on the content of a previous frame (and a dropped frame never arrives). This dependency means that streams with larger GOP sizes will, on average, be corrupted for a longer period of time when a frame is dropped.

In general, a larger GOP size reduces bandwidth demand, but the larger size also makes an encoded stream more vulnerable to network packet loss. A smaller GOP size increases the size of data transmissions and makes the encoded data more tolerant of packet loss. However, smaller GOP sizes increase encoding latency and bandwidth usage. Smaller GOP sizes create higher demands on GPU-device memory and system main memory.

In order to reduce latency while decreasing the disruption of visual stuttering, OpenUVR allows the MUD to send feedback to the host PC regarding dropped frames. When the MUD video decoder drops a frame because it hasn't received all of the data within a certain time limit, the MUD program triggers OpenUVRCP to send a short message to the host PC requesting that the host PC generate an I-frame on the next transmission. The MUD sends this control message after every frame until it receives an I-frame. When the host fulfills an I-frame request, it ignores additional requests from the MUD for a predetermined period of time to prevent the generation of consecutive large frames that would worsen network conditions.

### C. OpenUVR API

To enable OpenUVR in a VR application on the host PC, a programmer need only add a few function calls in the OpenUVR API or have a plug-in module invoke these API functions (if supported by the application framework). To collect inputs and display VR content, the system needs the MUD daemon to invoke the OpenUVR MUD API. Table I lists the basic functions necessary for the host PC and MUD to initiate OpenUVR in an application.

Figure 9(a) shows an example of code that uses these API functions in a VR application on the host PC. In the initialization section, the code creates buffer objects for OpenGL (or another graphics library) to store the generated content. After these buffer objects are created, the code uses the `openuvr_alloc_context` function to associate an OpenUVR context with a buffer object; the function also allows the programmer to specify both a video codec and a network protocol for carrying the VR content. Once the program has successfully set up an OpenUVR context, the program can set features for the context using `openuvr_set_feature`. The final portion of the code launches a thread that runs concurrently with the VR application to deliver content and handle signaling between the host PC and the MUD.

Figure 9(b) shows code that the MUD daemon runs. The daemon first initiates an `openuvr_mud_context` using the `openuvr_alloc_mud_context` function. Next, the program registers each input device (using the corresponding



Synopsis	Description
<b>OpenUVR Host API</b>	
struct openuvr_context *openuvr_alloc_context (enum OPENUVR_DECODER_TYPE dec_type, enum * OPENUVR_NETWORK_TYPE net_proto, void *ptr)	Initialize an OpenUVR context with the specified video encoder, the desired network protocol, and an OpenGL pixel buffer.
int openuvr_set_feature(struct openuvr_context *context, enum OPENUVR_FEATURE feature, int value)	Configure a feature of an existing openuvr_context with the specified value.
void openuvr_init_thread(struct openuvr_context *context)	Create an OpenUVR thread using the given OpenUVR context.
int openuvr_map_input_device(struct openuvr_mud_context *context, int id, int fd);	Map a device registered with id on the OpenUVR MUD device to a local device using its opened descriptor.
<b>OpenUVR MUD API</b>	
struct openuvr_mud_context *openuvr_alloc_mud_context (enum OPENUVR_DECODER_TYPE dec_type, enum OPENUVR_NETWORK_TYPE net_proto);	Create an OpenUVR MUD context with the specified decoder and network protocol.
int openuvr_init_mud_thread(struct openuvr_mud_context *context);	Create a MUD thread that captures inputs and renders the received VR content using a standard display device.
int openuvr_register_input_device(struct openuvr_mud_context *context, int id, int fd);	Register an input device to a device id in OpenUVR using an opened descriptor.

TABLE I  
THE OPENUVR API

```

GLuint pbo;
// generate an OpenGL buffer
glGenBuffers(1, &pbo);
// bind the created OpenGL buffer as a target to
//store generated content
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo);
// initializes the buffer object's data store
glBufferData(GL_PIXEL_PACK_BUFFER, 1920*1080*4, 0,
             GL_DYNAMIC_COPY);
// read a block of pixels from the frame buffer
glReadPixels(0, 0, 1920, 1080, GL_RGB, GL_UNSIGNED_BYTE, 0);
// initialize the OpenUVR context
ouvr_ctx = openuvr_alloc_context(OPENUVR_ENCODER_H264_CUDA,
                                OPENUVR_NETWORK_RAW, &pbo);

if(ouvr_ctx == NULL) {
    ri.Printf(PRINT_ALL, "couldn't allocate openuvr
context\n");
    exit(1);
}

// set the target FPS for OpenUVR
openuvr_set_feature(ouvr_ctx, TARGET_FPS, 60);
// map an OpenUVR MUD input to the OpenUVR context
int fd = open("/dev/input/mice", O_RDONLY);
openuvr_map_device(ouvr_ctx, SYS_MOUSE, fd);
// create an OpenUVR thread to send VR content and
// receive user inputs
openuvr_init_thread(ouvr_ctx);
(a)

// initialize the OpenUVR context
ouvr_ctx = openuvr_alloc_mud_context
            (OPENUVR_ENCODER_H264_CUDA,
             OPENUVR_NETWORK_RAW);

if(ouvr_ctx == NULL) {
    ri.Printf(PRINT_ALL, "couldn't allocate openuvr
context\n");
    exit(1);
}

// map an OpenUVR MUD input to the OpenUVR MUD context
int fd = open("/dev/input/mice", O_RDONLY);
openuvr_register_input_device(ouvr_ctx, SYS_MOUSE, fd);
// create an OpenUVR thread to send VR content and
// receive user inputs
openuvr_init_mud_thread(ouvr_ctx);
(b)

```

Fig. 9. The OpenUVR setup-initialization code in (a) the VR host application and (b) the MUD daemon

descriptor) and associates each device with an ID known by both the MUD and the host-PC application. The program then invokes the `openuvr_init_thread` function to create a thread that continually displays the received VR content and

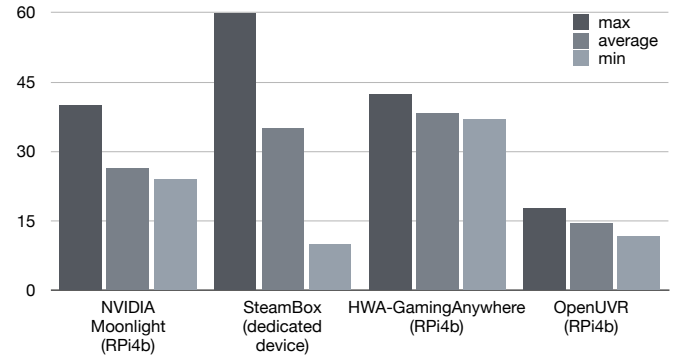


Fig. 10. The visual latencies of OpenUVR, HWA-GamingAnywhere, and other VR systems

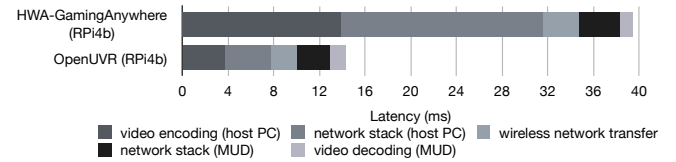


Fig. 11. The latency breakdown for HWA-GamingAnywhere and OpenUVR

We then used visual-latency performance assessments, a user study, and differing hardware configurations to evaluate the optimizations.

#### A. Latency

In this section, we evaluate the visual latency of OpenUVR and describe how each optimization helps reduce latency.

1) *Visual Latency*: Figure 10 compares the visual latency of OpenUVR with that of HWA-GamingAnywhere and other VR systems. We used exactly the same environment as in Section III-A. The average latency of OpenUVR is only 14.32 ms, which is  $2.67\times$  better than the average latency of HWA-GamingAnywhere. OpenUVR's latency is also lower than the latency of the 60-FPS display that an RPI4b can support. Of crucial importance is the fact that OpenUVR's latency falls below the threshold for motion-sickness onset.

2) *Latency reductions from optimizations*: Figure 11 compares the latency breakdown of OpenUVR with HWA-GamingAnywhere and Table II summarizes the effect of each optimization discussed in Section IV and shows how

Using the observations from Section III, we optimized the OpenUVR system stacks to further reduce end-to-end latency.

Optimization	Latency difference on Host PC ( $\mu$ s)	Latency difference on MUD ( $\mu$ s)	Latency difference on Network ( $\mu$ s)	Cumulative difference ( $\mu$ s)
Transcoding avoidance (RGB encoding)	-5,507	312	800	-4,395
Plug-in modules+ kernel-space functions	-4,710	-	-	-9,105
Direct network I/O	-13,672	-707	-	-23,484
P2P Network	-	-	-1,600	-25,084
Feedback control	-10	-40	-	-25,134
All optimizations	-23,899	-435	-800	-25,134

TABLE II

THE IMPROVEMENT ACHIEVED BY EACH CUMULATIVE OPENUVR OPTIMIZATION, COMPARED WITH HWA-GAMINGANYWHERE

OpenUVR brings the average visual delay down to less than one 60-FPS frame. The most significant reductions in latency were produced on the host PC, with minor improvements observed for the MUD. As Figure 11 depicts, OpenUVR spends only 3.72 ms in video encoding on the host PC, 3.95 ms on the network stack of the host PC, 2.4 ms on the network medium, 2.86 ms on the network stack of RPi4b and 1.39 ms on video decoding on RPi4b.

By avoiding the color-space conversion, OpenUVR removes 5.51 ms of end-to-end latency. Though using RGB encoding increases data size, the increase is offset by the P2P network topology, which reduces network-transfer latency by 0.8 ms. As a reference, with P2P and YUV encoding transfer latency reduced by 1.6 ms.

Data sharing between OpenUVR and the running application reduces end-to-end latency by 4.71 ms. The data sharing is made possible through use of the OpenUVR API for applications and the plug-in module for the underlying game engine). The direct-network I/O mechanism lowers network-stack latency by 13.67 ms on the host PC and 0.7 ms on the MUD. OpenUVRCP's feedback control improves overall latency by only 0.1 ms, but the resulting video quality is far superior to the quality achieved without feedback control.

### B. Energy consumption

We also evaluated the energy consumption of the OpenUVR system. We used a single USB power bank with a maximum capacity of 5,400 mAh to simultaneously supply the whole MUD, the RPi4b, and the HMD (with RPi4b output connected to the HMD). This single-power-supply setup allowed the MUD to run for 6 hours and 20 minutes. As a reference, a fully charged Oculus Quest running for 8 hours consumes 10,000 mAh —significantly more energy than OpenUVR needs [35].

### C. Performance sensitivity of GPU architectures

As the main computation resource for generating VR content and encoding H.264 data, the GPU plays an important role in a VR system. In addition to using the RTX 2080 GPU (mentioned in Section III-A) to analyze OpenUVR's performance, we used a mid-end NVIDIA GTX 1060 graphics card. The GTX 1060 has smaller amount of memory than the RTX 2080 (6 GB v.s. 8GB) and uses an earlier Pascal microarchitecture. Both cards run at about the same frequencies (1708 MHz vs.

longer encoding time that the GTX 1060 requires. However, even with the increased latency, OpenUVR still reduces the video encoding latency by 8 ms from purely relying on the computation power of CPUs. The performance of hardware-accelerated video encoding for modern GPU architectures thus has a significant impact on end-to-end latency in the current version of OpenUVR and we argue that hardware accelerated video encoding is necessary since future CPU architectures are unlikely to close the performance gap.

### D. Quality

To evaluate the quality of OpenUVR, we conducted two types of experiments. A quantitative analysis and a series of user study.

1) *Quantitative analysis*: To calculate the Structural Similarity (SSIM), a quantitative measurement of loss in frame quality between the original version and the user-perceived version that the video processing community uses [36], we instrumented code on both the OpenUVR content capturing module and the MUD module to record the received frames on both sides. The modified modules will record 150 successive frames for every 10 seconds on both the host PC and the MUD since the game starts. At the end of the experiment, we the average SSIM score for each batch of 150 frames and the average throughout the whole 10-minute experiment as Section III-A3.

OpenUVR obtains the best SSIM value for a batch at 0.961 and an average of 0.918 among all batches in our experiments. The average SSIM score indicates that OpenUVR achieves “good” visual quality that surpasses the 0.9 SSIM threshold [37]. The SSIM of the worst-case batch in our experiments is 0.679. The worst-case scenario occurs when the user avatar enters a new room that is significantly different from the previous room. We found that the frame rate of the host PC also drops at the same time, a potential source that affects the efficiency of OpenUVR host system modules and causes drops of the encoded frames.

2) *User study*: We also conducted a single-blind A/B user study to identify perceived differences between an OpenUVR experience and a native host-PC experience. We randomly selected one hundred college participants across different institutions. We asked the participants, aged 20–27, to play two sessions of Unreal Tournament. For session A, each user played on a direct HDMI connection to the host PC. For session B, each user played on an RPi4b using OpenUVR. Both connections were output to a single monitor. We used an HDMI switcher to facilitate switching between the two inputs and set the monitor resolutions to 1920x1080 for both sessions. Whether the user started with session A or B was randomized



pdfelement

試用版

by flipping a coin. After completing both sessions, we asked the user which session the user preferred in terms of latency and video quality.

Whereas 51% of users preferred the OpenUVR-based session on the RPi4b (session B), 49% preferred the session on the host PC (session A). Most users stated that they could not identify a difference between the two sessions, but 24% of the participants were adamant about either the host PC or the RPi4b providing a substantially better-quality gaming experience. These results suggest that there is no perceivable difference between the quality of a wired system and a well-functioning OpenUVR system.

#### E. Synchronous OpenUVR

In its default configuration, OpenUVR initiates an independent thread that uses a specific GPU-memory location to continually encode data and transmits the data to the MUD. This design means that OpenUVR's encoding and communication processes can pipeline with VR-content generation, improving the throughput of system resources.

The default OpenUVR configuration also allows the host to generate content at high frame rates (90 FPS for Unreal Tournament). But the Raspberry Pi MUD can only decode and display video at 60 FPS, so generating VR content at 90 FPS does not improve UX. An application that can render at 90 FPS takes at most 1/90th of a second, or 11.1 ms, to execute its game loop. If the application runs at 60 FPS instead, it has 16.7 ms to run its game loop, giving it 5.6 ms of slack time to perform OpenUVR's tasks, which only take around 3.72 ms.

OpenUVR can thus provide a synchronous API that a game can use to trigger encoding and data-exchange tasks. The lack of need for a separate thread also frees up the full CPU core, potentially lowering power consumption and/or the cost of the host PC. In our experiments, we did not observe any significant change in visual latency or quality due to OpenUVR's default configuration.

#### F. Discussion on future technologies

With transcoding avoidance, plug-in modules and implementing OpenUVR API backend as kernel-space functions, the remaining latency of OpenUVR starting from generating a picture frame before the encoded frame reaching the network stack is still 3.72 ms, accounting for 26% of the total end-to-end latency. The network stack on the host PC still consumes 28% of the total latency. Therefore, techniques or more advanced hardware accelerators (e.g., video codecs or network offloading engines) in reducing the computation/software overhead on the host PC are still effective as both accounts for more than half of the latency.

High bandwidth wireless network technologies can potentially offset video encoding latency only if the bandwidth can carry uncompressed raw content with very low latency. However, as the volume of a raw frame (i.e., 5.93 MB in full HD resolution) is significantly larger than an encoded frame (i.e., 0.5 MB in full HD resolution using H.264), the content also increases the latency in the system interconnect as well as the whole datapath on both the host PC and the MUD. In our experimental platform, copying a raw full HD frame from the GPU to the main memory takes

1.1 ms while an encoded frame takes only 250  $\mu$ s. We expect the similar amount of latency would occur when copying the frame to the wireless NIC as well as receiving the frame from the wireless NIC on the MUD side. Even without considering the buffering and computation overhead that next generation wireless technologies (e.g., mmWave and IEEE 802.11ad) typically incur, the aggregated additional latency due to the increased frame size is already near the amount of the latency of encoding and decoding the frame. Therefore, we expect techniques that OpenUVR exploits are still necessary even when the next generation wireless network technologies are presented.

## VI. RELATED WORK

As we discussed in Section III, the two major bottlenecks of datapath latency are the video codec and the network software stack. To reduce video latency, parallel video codecs [38], [39], [40], [41] divide an image into multiple segments to allow parallel encoding/decoding. Furion [2] goes further by examining the characteristics of VR content to achieve segmentation more efficiently. OpenUVR's latency reduction techniques, which avoid transcoding and virtual-memory overhead, complement parallel video codecs and Furion's enhancements of them. In addition, OpenUVR reduces computational and energy requirements for VR applications.

This paper did not explore optimizations for VR-content generation because other researchers have already succeeded in this area; however, the parallelization strategies of VR graphics engines do offer opportunities for improvement, notably with respect to picture-frame tiling. The graphics pipeline can be optimized by detecting redundant, identical tiles [42], [43], [44], by separating stable objects from frequently updated objects [45], [46], or by prioritizing image areas that attract user attention [47], [48]. Rollback-free value prediction [49], Potluck [50], AxGames [51], and PATU [52] apply approximate computing techniques to image elements that do not significantly affect UX. OpenUVR can work with any of these frameworks to reduce the encoded video-stream volume.

Slow-motion VR applications benefit from strategies that use free bandwidth to aggressively predict, precompute, or cache all possible images that a VR user might encounter. As these strategies are hardware-independent, many VR systems use them to improve UX. Examples of such systems include FlashBack [3], CoIC [53], Coterie [54], DeltaVR [55], MUVIR [56], and the system developed by Mahzari et. al [57]. Nonetheless, for fast-moving real-time VR games, such strategies may not work well since the user's motions continually change. Problems are especially acute for VR games with multiple players.

Placing computation-intensive tasks in the cloud provides an alternative VR architecture for thin user clients [58], [59], [60], [61], [28]. VisualCloud [58] tries to predict the user's orientation in order to reduce the latency associated with getting data from the cloud, and VisualCloud reduces bandwidth demand by appending video segments to the media stream. Even this combined approach isn't enough to overcome the long propagation delay between cloud servers and the user device, so state-of-the-art solutions still suffer from more than a 70 ms delay (and Outatime is no exception). That being said,



some promise may be found among VR applications that use edge nodes on 5G networks [62].

As network latency creates serious limitations for cloud-based solutions, VR systems using edge-computing architectures are attracting more attention [62], [63], [64], [65], [66]. But these architectures need to distribute workloads between cloud and edge nodes, which creates task-scheduling issues. Although communication-constrained mobile-edge computing (MEC) [63] optimizes task scheduling to minimize high communication costs and maximize system cache utilization, and although MSVR [64] hides the long network round-trip latency by processing delay-sensitive actions on the mobile device, both architectures require the client or HMD to have a certain degree of computation capability, which complicates application design. (Note that Kahawai [37] presents a novel way to distribute tasks between the host PC and mobile devices in order to accelerate graphical rendering and reduce bandwidth.) Since OpenUVR leaves a relatively small workload to the client and uses a more powerful local host PC to generate VR content, OpenUVR's architecture better addresses the network delay issue while keeping application design simple. The demands that OpenUVR places on the MUD are also the lowest among these frameworks.

OpenUVR builds upon previous research to bypass operating-system overhead and avoid unnecessary memory copies [67], [68], [69], [70], [44], [49] and adopts a minimum layered datapath design as IO-Lite [71]. Accomplishments from GPUnet can also help reduce latencies further [72], but GPUnet's reliance on RDMA hardware support in wireless NICs limits GPUnet's usefulness when it is used with existing commodity hardware. Without OpenUVR's full-stack design, GPUnet's optimizations can only impact the performance of a single system component and so fail to eliminate functionality in each stack layer. We expect OpenUVR's latency reduction techniques would apply to other latency-sensitive systems sharing similar issues [73].

OpenUVR's network stack resembles user-space Wi-Fi [74] in that the stack exposes the device layer directly to the application. OpenUVR adds a kernel module alongside the existing network interface. This feature allows conventional latency-insensitive applications to access the network stack and benefit from optimizations in each protocol; only OpenUVRDPP traffic goes through the direct network-access feature. Note that OpenUVR preserves access to the network-device buffer as a protected memory address in user-space and requires the user program to grant such access. Consequently, OpenUVR does not create security issues in VR systems.

Much potential still exists for architectural and system-level optimizations in VR gaming applications. Atomic Quake, which demonstrates the usefulness of transactional memory, serves as one such example [75], [76]. Aside from Atomic Quake, the previously mentioned graphical pipeline optimizations, and OpenUVR, we have found only limited discussion of how to modify computer architectures to enhance the performance of a complete VR system.

## VII. CONCLUSION

Due to severe constraints in latency and bandwidth, modern UVR systems do not provide excellent UX. In addition, exist-

ing VR-related research projects only take single-component approaches to optimizing system design, which limits the effectiveness of performance improvement. Moreover, a cross-layer design is difficult to implement because many existing VR systems use proprietary software/hardware components with only partially accessible designs, thereby discouraging or preventing the research community from completing a thorough examination of whole systems.

This paper presents OpenUVR, the first full-system, open-source UVR framework. OpenUVR works on top of commodity hardware components, making it easy to replicate the OpenUVR system and study the impact of each architectural element on the performance metrics to which VR applications are sensitive. OpenUVR also allows the research community to revisit and evaluate the interactions among different architectural and system components. Perhaps most importantly, the UX resulting from OpenUVR's optimized system architecture is better than those of existing commercialized products.

Our research shows that (1) a significant amount of latency resides in each layer of a modern VR system and (2) optimizing system software stacks and efficiently using architectural components can improve end-to-end latency to 14.32 ms, which is imperceptible to the user. We hope that OpenUVR's novel open-source design will enable more research in VR system performance and modification.

## REFERENCES

- [1] Y. Bao, H. Wu, T. Zhang, A. A. Ramli, and X. Liu, "Shooting a moving target: Motion-prediction-based transmission for 360-degree videos," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 1161–1170, Dec 2016.
- [2] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai, "Furion: Engineering high-quality immersive virtual reality on today's mobile devices," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, (New York, NY, USA), pp. 409–421, ACM, 2017.
- [3] K. Boos, D. Chu, and E. Cuervo, "FlashBack: Immersive virtual reality on mobile devices via rendering memoization," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, (New York, NY, USA), pp. 291–304, ACM, 2016.
- [4] T. A. Stoffregen, E. Faugloire, K. Yoshida, M. B. Flanagan, and O. Merhi, "Motion sickness and postural sway in console video games," *Human Factors*, vol. 50, no. 2, pp. 322–331, 2008. PMID: 18516842.
- [5] T. J. Buker, D. A. Vincenzi, and J. E. Deaton, "The effect of apparent latency on simulator sickness while using a see-through helmet-mounted display: Reducing apparent latency with predictive compensation," *Human Factors*, vol. 54, no. 2, pp. 235–249, 2012. PMID: 22624290.
- [6] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler, "Toward low-latency and ultra-reliable virtual reality," *IEEE Network*, vol. 32, pp. 78–84, March 2018.
- [7] Michael Abrash, "What VR could, should, and almost certainly will be within two years." <http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf>, 2014.
- [8] S. Davis, K. Nesbitt, and E. Nalivaiko, "A systematic review of cybersickness," in *Proceedings of the 2014 Conference on Interactive Entertainment*, IE2014, (New York, NY, USA), pp. 1–9, Association for Computing Machinery, 2014.
- [9] K. Raaen and I. Kjellmo, "Measuring latency in virtual reality systems," pp. 457–462, 09 2015.
- [10] E. Cuervo, "Beyond reality: Head-mounted displays for mobile systems researchers," *GetMobile: Mobile Comp. and Comm.*, vol. 21, pp. 9–15, Aug. 2017.
- [11] D. Kushner, "Virtual reality's moment," *IEEE Spectrum*, vol. 51, pp. 34–37, January 2014.
- [12] HTC Corporation, "VIVE." <https://www.vive.com/us/>.
- [13] Google, "Daydream." <https://vr.google.com/daydream/>.
- [14] L. Samsung ELECTRONICS CO., "Gear VR." <https://www.samsung.com/global/galaxy/gear-vr/>.



- of the 2017 ACM International Conference on Management of Data, SIGMOD '17, (New York, NY, USA), pp. 1615–1618, ACM, 2017.
- [59] G. Illahi, M. Siekkinen, and E. Masala, “Foveated video streaming for cloud gaming,” *CoRR*, vol. abs/1706.04804, 2017.
  - [60] T. Kämäräinen, M. Siekkinen, A. Ylä-Jääski, W. Zhang, and P. Hui, “A measurement study on achieving imperceptible latency in mobile cloud gaming,” in *Proceedings of the 8th ACM on Multimedia Systems Conference*, MMSys'17, (New York, NY, USA), pp. 88–99, ACM, 2017.
  - [61] K. Lee, D. Chu, E. C. Laffaye, 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 *MobiSys*, 2015.
  - [62] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler, “Toward low-latency and ultra-reliable virtual reality,” *IEEE Network*, vol. 32, pp. 78–84, March 2018.
  - [63] X. Yang, Z. Chen, K. Li, Y. Sun, N. Liu, W. Xie, and Y. Zhao, “Communication-constrained mobile edge computing systems for wireless virtual reality: Scheduling and tradeoff,” *IEEE Access*, vol. 6, pp. 16665–16677, 2018.
  - [64] L. Zhang, L. Sun, W. Wang, and J. Liu, “Unlocking the door to mobile social VR: Architecture, experiments and challenges,” *IEEE Network*, vol. 32, pp. 160–165, Jan 2018.
  - [65] Y. Leng, C.-C. Chen, Q. Sun, J. Huang, and Y. Zhu, “Energy-efficient video processing for virtual reality,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), pp. 91 —103, Association for Computing Machinery, 2019.
  - [66] S. Shi, V. Gupta, M. Hwang, and R. Jana, “Mobile VR on edge cloud: A latency-driven design,” in *Proceedings of the 10th ACM Multimedia Systems Conference*, MMSys '19, (New York, NY, USA), pp. 222 —231, Association for Computing Machinery, 2019.
  - [67] H.-k. J. Chu, “Zero-copy TCP in Solaris,” in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, (USA), p. 21, USENIX Association, 1996.
  - [68] Y. A. Khalidi, J. M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani, “Solaris MC: A multi computer OS,” in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, (USA), p. 16, USENIX Association, 1996.
  - [69] S. Kato, J. Aumiller, and S. Brandt, “Zero-copy I/O processing for low-latency GPU computing,” in *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pp. 170–178, April 2013.
  - [70] A. Maghazeh, U. D. Bordoloi, M. Villani, P. Eles, and Z. Peng, “Perception-aware power management for mobile games via dynamic resolution scaling,” in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 613–620, Nov 2015.
  - [71] V. S. Pai, P. Druschel, and W. Zwaenepoel, “Io-lite: A unified i/o buffering and caching system,” *ACM Trans. Comput. Syst.*, vol. 18, no. 1, pp. 37–66, 2000.
  - [72] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, “GPUnet: Networking abstractions for GPU programs,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 201–216, USENIX Association, 2014.
  - [73] R. Liu and F. X. Lin, “Understanding the characteristics of android wear os,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pp. 151–164, 2016.
  - [74] M. Backhaus, M. Theil, M. Rossberg, and G. Schaefer, “Towards a flexible user-space architecture for high-performance IEEE 802.11 processing,” in *2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 1–9, Oct 2018.
  - [75] F. Zylkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, “Atomic Quake: Using transactional memory in an interactive multiplayer game server,” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '09, (New York, NY, USA), pp. 25–34, ACM, 2009.
  - [76] V. Gajinov, F. Zylkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero, “QuakeTM: Parallelizing a complex sequential application using transactional memory,” in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 126–135, ACM, 2009.