Cutting the Cord: Designing a High-quality Untethered VR System with Low Latency Remote Rendering

Conference Paper · June 2018

DOI: 10.1145/3210240.3210313

7 authors, including:

Luyang Liu
Rutgers, The State University of New Jersey
10 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Wuyang Zhang
Rutgers, The State University of New Jersey
3 PUBLICATIONS **9** CITATIONS
SEE PROFILE

Marco Gruteser
Rutgers, The State University of New Jersey
191 PUBLICATIONS **6,969** CITATIONS
SEE PROFILE

Cutting the Cord: Designing a High-quality Untethered VR System with Low Latency Remote Rendering

Luyang Liu[†], Ruiguang Zhong[§], Wuyang Zhang[†], Yunxin Liu[*],
Jiansong Zhang[♯], Lintao Zhang[*], Marco Gruteser[†]

[†]WINLAB, Rutgers University
{luyang, wuyang, gruteser}@winlab.rutgers.edu
[*]Microsoft Research
{yunxin.liu, lintaoz}@microsoft.com

[§]Beijing University of Posts and Telecommunications
bu@bupt.edu.cn
[♯]Alibaba Group
muduan.zjs@alibaba-inc.com

ABSTRACT

This paper introduces an end-to-end untethered VR system design and open platform that can meet virtual reality latency and quality requirements at 4K resolution over a wireless link. High-quality VR systems generate graphics data at a data rate much higher than those supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. The necessary image encoding, makes it challenging to maintain the stringent VR latency requirements. To achieve the required latency, our system employs a Parallel Rendering and Streaming mechanism to reduce the add-on streaming latency, by pipelining the rendering, encoding, transmission and decoding procedures. Furthermore, we introduce a Remote VSync Driven Rendering technique to minimize display latency. To evaluate the system, we implement an end-to-end remote rendering platform on commodity hardware over a 60Ghz wireless network. Results show that the system can support current 2160x1200 VR resolution at 90Hz with less than 16ms end-to-end latency, and 4K resolution with 20ms latency, while keeping a visually lossless image quality to the user.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing systems and tools**; • **Computer systems organization** → *Real-time system architecture*;

KEYWORDS

Virtual Reality, Low latency, Untethered, High-quality, 60GHz

ACM Reference Format:
Luyang Liu[†], Ruiguang Zhong[§], Wuyang Zhang[†], Yunxin Liu[*], Jiansong Zhang[♯], Lintao Zhang[*], Marco Gruteser[†]. 2018. Cutting the Cord: Designing a High-quality Untethered VR System with Low Latency Remote Rendering. In *Proceedings of MobiSys'18*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3210240.3210313

1 INTRODUCTION

Virtual reality systems have provided unprecedented immersive experiences in the fields of video gaming, education, and healthcare. Reports forecast that 99 million Virtual Reality (VR) and Augmented Reality (AR) devices will be shipped in 2021 [1], and that the market will reach 108 billion dollars [2] by then. Virtual and augmented reality is also a key application driver for edge-computing and high-bandwidth wireless networking research.

Existing VR systems can be divided into two categories: High-quality VR and standalone VR systems. Due to the requirements of high quality and low latency, most high-quality VR systems, such as HTC Vive [3] and Oculus Rift [4], leverage a powerful desktop PC to render rich graphics contents at high frame rates and visual quality. However, most of these solutions are *tethered*: they must be connected to a PC via a USB cable for sending sensor data from the Head Mounted Display (HMD) to the PC and an HDMI cable for sending graphics contents from the PC back to the HMD. These cables not only limit the user's mobility but also impose hazards such as a user tripping or wrapping the cable around the neck. Standalone, portable VR systems such as Samsung Gear VR [5] and Google Daydream [6] run VR apps and render graphics contents locally on the headset (or a smartphone slide in the headset). Those VR systems allow untethered use but the rendering quality is limited by the capability of the headset or smartphone. There has been extensive demand for such untethered high-quality VR systems.

Untethered high-quality VR is highly desirable but extremely challenging. Ideally, the cables between the VR HMD and PC should be replaced by a wireless link. However, even existing high-quality VR systems operate at 2160x1200 resolution and 90Hz, which generates a data rate much higher than those supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. The necessary image encoding, makes it difficult to maintain the stringent VR motion-to-photon latency requirements, which are necessary to reduce motion sickness.

VR applications have motivated much wireless research to realize robust, high-capacity connectivity, for example in the 60 GHz range. Most existing research has independently focused on optimizing the wireless link [7–10] or the VR graphics pipeline [11, 12].

MobiSys'18, June 10–15, 2018, Munich, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5720-3...$15.00
https://doi.org/10.1145/3210240.3210313

To enable high-quality VR on smartphones, Furion [11] separates the rendering pipeline into tasks to render the image foreground and tasks to render the image background, so that the background rendering tasks can be offloaded over commodity Wi-Fi. Other emerging systems such as TPCAST [13] and DisplayLink [14] replace the HDMI cable with a wireless link to enable untethered VR experience with remote rendering and streaming. However, none of them studied systems issues and optimization opportunities that arise when combining rendering, streaming, and display. Furthermore, it is still challenging to enable untethered VR for future 4K or 8K systems with framerates larger than 90Hz. Additionally, we discover that displaying remote-rendered frames on an HMD may also introduce extra latency due to the VSync driven rendering and display policy.

To overcome these challenges and facilitate such research, we propose an open remote rendering platform that can enable high-quality untethered VR with low latency on general purpose PC hardware. It reduces the streaming latency caused by frame rendering, encoding, transmitting, decoding and display through a Parallel Rendering and Streaming Pipeline (PRS) and Remote VSync Driven Frame Rendering (RVDR). PRS pipelines the rendering, encoding, transmission, and decoding process. It also parallelizes the frame encoding process on GPU hardware encoders. The RVDR technique carefully schedules the start time of sensor acquisition and rendering new frames on the server side so that the result arrives at the display just before the VSync screen update signal, thus reducing latency caused by the display update.

To evaluate the system, we implemented the end-to-end remote rendering platform on commodity hardware over a 60GHz wireless network. The result shows that the system supports existing high-quality VR graphics with a latency of less than 16ms. We further show promise to support future VR systems with 4K resolutions with a latency up to 20ms. To facilitate widespread use of this platform, it is currently designed as a software system using only general-purpose GPU and network hardware. Performance could also be further improved through hardware implementations of key components.

The contributions of this work are:

- Quantifying component latency in an end-to-end wireless VR system and identifying the impact of the screen refresh (VSync). §2
- Designing a pipeline of Parallel Rendering and Streaming to reduce the streaming latency caused by encoding, transmitting and decoding the frames. §4
- Developing a method of Remote VSync Driven Rendering to adjust the timing of sensor data acquisition and rendering of a new frame based on the screen refresh timing (VSync signal) on the client side. §5
- Implementing and evaluating an open end-to-end remote rendering and streaming platform based on the system on commodity hardware over a 60GHz wireless network. §6
- Showing that the platform can support the current 2160x1200 VR resolution and the 4K resolution at 90Hz within 20ms latency over a stable 60GHz link even without complex modifications of the rendering process. §7

2 CHALLENGES AND LATENCY ANALYSIS

Designing a high-quality untethered VR system is extremely challenging due to the stringent requirements on data throughput and end-to-end latency. Assuming we use three bytes to encode the RGB data of each pixel, for HTC Vive and Oculus Rift with a frame rate of 90Hz and a resolution of 2160x1200, the raw data rate is 5.6Gbps, much higher than the data rate (e.g., less than 2Gbps) supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. For future VR targeting at a resolution of 4K UHD or even 8K UHD, the required data rate would be as high as 17.9Gbps and 71.7Gbps, respectively.

To address the challenge of high data throughput, data compression is necessary. However, high-quality VR also requires a very tight total end-to-end (i.e., motion-to-photon) latency of 20-25ms [12] to reduce motion sickness. That is, once the HMD moves, the system must be able to display a new frame generated from the new pose of the HMD within 20-25ms. As compressing and decompressing frames introduce extra latency, it is even more challenging to meet the end-to-end latency requirement.

Latency analysis. We use the following equations to model the end-to-end latency of our proposed untethered VR system with remote rendering:

$$T_{e2e} = T_{sense} + T_{render} + T_{stream} + T_{display} \tag{1}$$

$$T_{stream} = T_{encode} + T_{trans} + T_{decode} \tag{2}$$

$$T_{trans} = \frac{FrameSize}{Throughput} \tag{3}$$

T_{e2e} is the total end-to-end latency in generating and displaying a new frame. It consists of four parts: the time for the rendering server to retrieve sensor data from the HMD (T_{sense}); the time for the rendering server to generate a new frame (T_{render}); the time to send the new frame from the rendering server to the HMD (T_{stream}); and the time for the HMD to display the new frame ($T_{display}$).

T_{stream} is the extra latency introduced by cutting the cord of a tethered VR system. It has three parts: the time to compress a frame on the rendering server (T_{encode}); the time to transmit the compressed frame from the rendering server to the HMD over a wireless connection (T_{trans}); and the time to decompress the received frame on the HMD (T_{decode}). T_{trans} is decided by the compressed frame size and the data throughput of the wireless connection.

$T_{display}$ also introduces significant latency. In modern graphics systems, frame displaying is driven by VSync signals that are periodically generated based on the screen refreshing rate. If a frame misses the current VSync signal after it is received and decoded on an HMD, it must wait in the frame buffer for the next VSync signal before it can actually be displayed on the screen (see more details in §5). For a frame rate of 90Hz, the average waiting time is 5.5ms. Such an extra latency may significantly impact the performance of a high-quality untethered VR system, and thus must be carefully mitigated as much as possible.

For the total 20-25ms budget of T_{e2e}, T_{sense} is small (less than 400μs in our system with a WiGig network). T_{render} may be 5-11ms

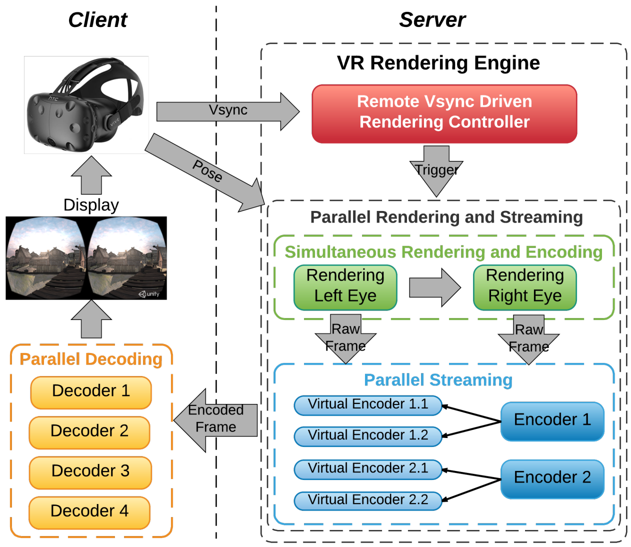


Figure 1: System architecture.

depending on the rendering load. With a $T_{display}$ of 5.5ms, we have less than 10ms left for T_{stream}, including encoding, transmitting and decoding a frame, which makes it a very challenging task to meet the latency requirement of high-quality VR.

In this paper, we focus on minimizing T_{stream} to reduce the end-to-end latency T_{e2e}. This is mainly done through parallelizing frame rendering and frame encoding by leveraging the hardware capability. We cannot change T_{sense} and T_{render}. However, by carefully arranging the timing of rendering new frames with the latest pose of the HMD, we are able to reduce the frame waiting time for VSync signals, and thus reduce $T_{display}$.

Next, we describe our system design and the key techniques on how to address the challenges.

3 SYSTEM OVERVIEW

Figure 1 shows the system architecture of our proposed untethered VR system with remote rendering. At a high level, it has two parts connected through a wireless link: an HMD as the client and a PC as the rendering server. The HMD client tracks the pose of the player and the timing of its VSync signals and sends the recorded data to the rendering server. If the player uses extra controllers to play the game, the client also sends the controller data to the server. Using the data received from the client, the server renders new frames, compresses and transmits them to the client. Upon receiving a new frame, the client decompresses and displays it on the HMD. The wireless link can be WiFi or 60GHz wireless communication such as WiGig. In our implementation, we use WiGig for its high data throughput and low latency.

To reduce the latency of streaming frames from the rendering server to the HMD client, we develop two key techniques. The first technique is *Parallel Rendering and Streaming* (PRS). PRS takes advantage of the two-eye image rendering nature in VR. Once the left eye image is rendered, PRS immediately sends it to the hardware encoder for compression. At the same time, PRS continues to render the right eye image, enabling simultaneous rendering and encoding. PRS further divides the image of each eye into two slides for parallel four-way encoding to fully utilize the hardware encoding capability. After a frame slide is encoded, it is immediately sent to the wireless link for transmission, without waiting for the whole frame to be compressed. Similarly, once the HMD receives a frame slide, it also immediately starts to decompress it, without waiting for the other frame slides. Consequently, we achieve parallel frame rendering, encoding, transmission, and decoding, and thus significantly reduce the latency.

The second technique is *Remote VSync Driven Rendering* (RVDR). The key idea of RVDR is to reduce display latency by deciding when to acquire head tracking sensor data and render a new frame based on the timing of the VSync signals of the HMD client. To do so, the client keeps tracking the time of its last VSync signal and the display delay of the last frame. Based on these timing information, the rendering server decides whether to start earlier to render the next frame so that the next frame can meet the next VSync signal on the client, or to slightly postpone the sensor acquisition and rendering of the next frame so that it arrives closer to the VSync signal and display latency is reduced. This is effective because the frame can be rendered with the latest possible pose of the HMD and thereby reduce motion-to-photon latency. As a result, we further reduce the display latency and minimize the rate of missing frames to maximize the user experience.

In our design, the HMD client uses dedicated hardware video codecs (e.g., H.264) to decode the frames rendered on the rendering server. We choose this design because hardware video codecs have a small size and consume a low power. Prior studies [11, 15] have shown that it is not practically feasible to decode frames using CPU or GPU on an HMD, due to the high decoding latency and high power consumption. Hardware video codecs are mature and cost-effective techniques, widely used in smartphones, tablets, laptops and many other devices. Thus, integrating hardware video codecs into HMDs is a practical solution to enable high-quality untethered VR systems.

4 PARALLEL RENDERING AND STREAMING PIPELINE

In this section, we introduce how we use the *Parallel Rendering and Streaming* (PRS) mechanism to minimize the streaming latency (T_{stream}) on commercial VR platforms. PRS consists of two parts: *simultaneous rendering and encoding*, and *parallel streaming*. Together, they build a parallel frame rendering and streaming pipeline to reduce the streaming latency.

4.1 Simultaneous Rendering and Encoding

Rendering a frame with rich graphics contents may take a long time (e.g., longer than 5ms) even on a powerful VR-ready GPU (e.g., Nvidia Titan X). As we cannot simply reduce frame quality to save frame-rendering time, we must find other ways to mitigate the impact of the long frame-rendering time on the end-to-end latency. To this end, we propose the approach of simultaneous rendering and encoding, to allow starting to encode a frame while it is still being rendered.

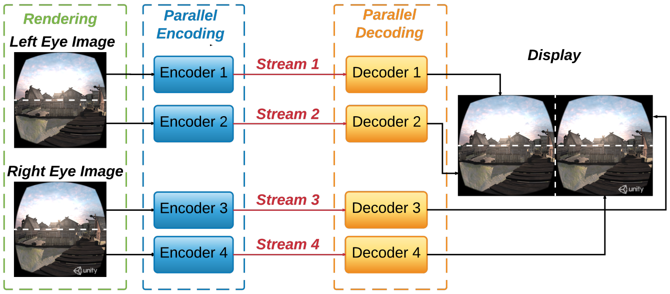


Figure 2: Simultaneous rendering and encoding with 4-way parallel streaming.

Simultaneous rendering and encoding is feasible due to two reasons. First, we observe that rendering a VR frame is typically done in three sequential steps: (1) render the left eye image, (2) render the right eye image, and (3) apply lens distortion on the whole frame so that the frame can be correctly displayed on a VR headset. This sequential rendering of the two-eye images provides an opportunity for us to start to encode the left eye image before the right eye image is fully rendered. Second, modern GPUs have dedicated hardware encoders and decoders that are separate from the GPU cores used for rendering VR frames (e.g., CUDA cores in Nvidia GPUs). Therefore, we may leverage the dedicated hardware encoders to compress a frame without impacting the performance of VR rendering.

Specifically, in simultaneous rendering and encoding, we redesign the VR rendering procedure to the following 6 steps: (1) render the left eye image, (2) apply lens distortion on the left eye image, (3) pass the distorted left eye image to the encoding pipeline in a separate thread, and at the same time (4) render the right eye image, (5) apply lens distortion on the right eye image, (6) pass the distorted right eye image to the encoding pipeline in another separate thread. Note that only steps (1), (2), (4), and (5) execute on the main rendering thread, while steps (3) and (6) execute on two separate encoding threads using hardware-based encoders. These encoding operations mainly consume the hardware-based encoder resources with light CPU usage, and thus do not block or slow down the frame-rendering pipeline inside the GPU.

Traditional video streaming approaches usually also use hardware-based encoders to accelerate the video-encoding procedure. However, they keep waiting for a frame being fully rendered before passing the entire frame to a hardware encoder. Such a design largely increases the end-to-end latency, which is fine to video streaming with a low frame rate (e.g., 30 fps) and without user interactions, but it is not acceptable in high-quality interactive VR systems.

4.2 Parallel Streaming

To further reduce the streaming latency, we propose to use a multi-threaded streaming technique to encode the image of each eye in multiple encoding threads. This is because almost all high-performance GPUs support more than one video encoding session [16] and each encoding session generates its own encoding stream independently.

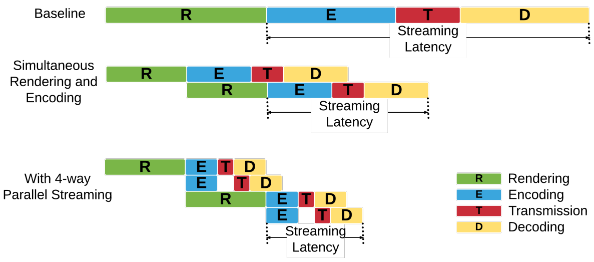


Figure 3: Illustration of parallel rendering and streaming (PRS) pipeline.

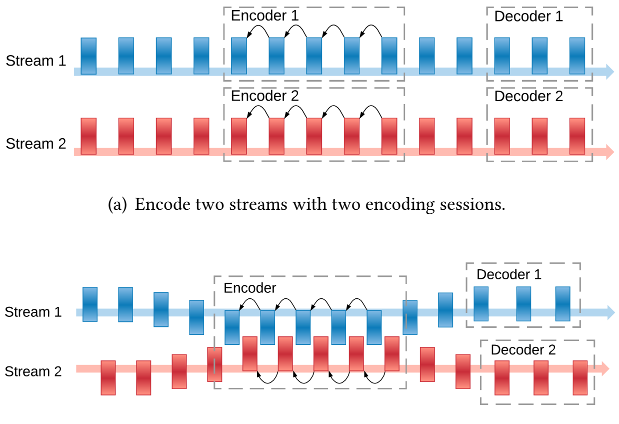
Consequently, by dividing an image into multiple slides and encoding each slide using different encoding sessions in parallel, we can reduce the total encoding time. In our system, we cut the image of each eye into two slides and compress each slide in a separate video stream. In total, we have four slides of the two eyes for 4-way parallel streaming. On the client side, multiple decoding sessions are used to decode each image slide in parallel as well.

This *parallel streaming* mechanism can be combined with *simultaneous rendering and encoding*. Figure 2 shows the process of simultaneous rendering and encoding together with 4-way parallel streaming. The image of each eye is divided into two slides: the upper one and the bottom one. The total four image slides are encoded into four video streams using four encoders. Accordingly, the HMD client uses four decoders to decode the four video streams, composites the four image slides into a full frame, and displays the frame on the HMD.

Figure 3 illustrates how the PRS mechanism can reduce the streaming latency through simultaneous rendering and encoding and 4-way parallel streaming, in comparison to a baseline approach. Four main tasks (rendering, encoding, transmission, and decoding) are represented with rectangles in different colors. The length of each rectangle is the rough execution time of the corresponding task. Note that here we analyze only the streaming latency rather the total end-to-end latency. Thus, we do not show the time of fetching the sensor data before rendering a frame and the time that the frame waits in the frame buffer after it is decoded but before it is displayed. In the baseline approach, the four tasks execute sequentially. The streaming latency, as shown in Figure 3, is the total execution time of encoding, transmitting and decoding the whole frame.

With *simultaneous rendering and encoding* (middle in Figure 3), the encoding of the left eye image starts immediately after the left eye image was rendering and in parallel with the rendering of the right eye image. As a result, this two-way parallel approach reduces the user-perceivable *add-on streaming latency*, i.e., the extra streaming latency after the whole frame is rendered, by 1/2. By combining the *simultaneous rendering and encoding* with 4-way *parallel streaming* together (bottom in Figure 3), we use two encoders to compress the image of one eye in parallel. The add-on streaming latency further reduces to around 1/4 of the one in baseline approach.

The parallel streaming technique may be further extended to 8-way or even 16-way for more parallelisms. However, we do not recommend doing so because 1) it requires more simultaneous



(a) Encode two streams with two encoding sessions.

(b) Encode two streams with only one encoding session.

Figure 4: Encoder multiplexing.

encoding sessions that may not be possible on many GPUs as we will show later, 2) it reduces the performance of motion estimation in H.264 and thus leads to a lower compression rate, and 3) it makes the implementation more complex.

4.2.1 Encoder Multiplexing. Ideally, the 4-way parallel streaming approach requires four encoding sessions. However, in practice, many GPUs including some popular commercial VR-ready GPUs may not support four simultaneous encoding sessions. For example, the Nvidia's GeForce 9 and 10 series and TITAN X GPUs [17] can support maximum two encoding sessions running simultaneously on a single GPU [1]. Consequently, the 4-way parallel streaming approach cannot directly work on those GPUs.

From Figure 3, we observe that even though the 4-way parallel streaming approach uses four encoding streams, there are only up to two streams overlapped at any time. This is because the rendering of the left eye image and the rendering of the right eye image are sequential. As rendering an image usually takes a longer time than compressing the image, encoding the half image of the left eye is expected to be much faster than rendering the image of the right eye. Therefore, only the encoding of the upper and bottom slides of the same eye image overlap. That is, only two encoding sessions are actually needed at the same time.

However, we still cannot directly encode four streams using only two encoding sessions due to the inter-frame compression in video encoding (e.g., H.264 and H.265). Compared to image compression that compresses independent images (e.g., JPEG), video compression encodes a set of images into a video stream containing 3 types of frames: I-frames, P-frames, and B-frames. An I-frame is encoded from a single image and can be decoded independently from other frames. However, a P-frame contains references from the previously encoded frame for a higher compression ratio, and therefore cannot be decoded by itself. A B-frame further uses bi-directional prediction that requires both its prior frame and its latter frame as its references, introducing more inter-frame dependencies in its encoding and decoding [2]. These inter-frame dependencies make it hard to encode two video sources using one single encoder.

To solve the problem, we propose *encoder multiplexing* [3] to temporally allow two video streams to share the same encoding session, as shown in Figure 4. Figure 4(a) shows the standard usage of a single encoding session for a single video stream. The two video streams are encoded in two different encoding sessions separately. Each encoder compresses its raw input frames rendered from the rendering server into an encoded H.264 stream, and sends the encoded stream through the network link to the client. Each P-frame references to its previous frame in the same stream and thus can be correctly decoded by the corresponding decoder.

As shown in Figure 4, with *encoder multiplexing*, we encode the two video streams in the same encoding session. To encode each P-frame correctly, we set the previous frame in the same video stream as the long-term reference frame (LRF) of the P-frame. As a result, each P-frame references to the previous frame in its own video stream rather than the previous frame that is encoded in the encoding session (which is from the other stream). The outputs of the encoder will be divided into two streams that are sent to two different decoders on the client. Even though each decoder only receives half of the encoded outputs of the encoder, it has the reference frame needed to decode received P-frames. What we need to do is only changing the list of decoded picture buffer (DPB) before passing each stream to the decoder, to let it ignore those missed frames (i.e., the frames of the other steam).

Specifically, we use one encoding session to encode the upper half images of two eyes and another encoding session to encode the bottom half images of two eyes, respectively. As a result, we enable four virtual encoders for the 4-way parallel streaming using only two encoding sessions and make our approach work on most GPUs.

5 REMOTE VSYNC DRIVEN RENDERING

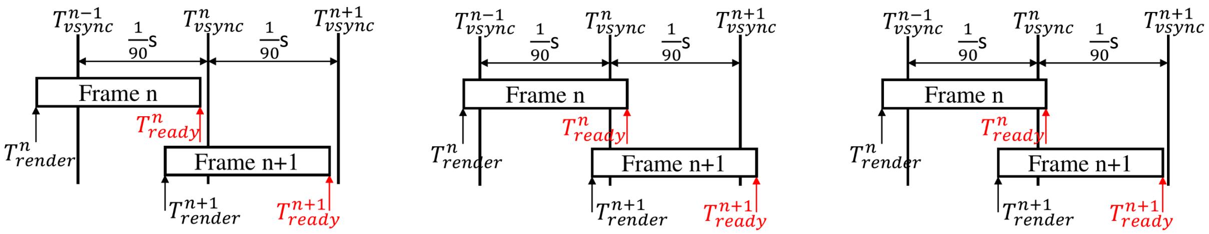
Modern computer systems use VSync (Vertical Synchronization) signals to synchronize the rate of rendering frames (i.e., frame rate) and the refresh rate of a display. To ensure a smooth user experience (e.g., avoid screen tearing), the *double buffering* technique is usually used with two frame buffers: a front frame buffer containing the frame that is being displayed on the screen, and a back frame buffer containing the frame that is being rendered. Upon receiving a VSync signal, the system swaps the two buffers to display the newly rendered frame and continues to render the next frame in the new back frame buffer. If the system renders a frame very fast, the frame must wait for the next VSync signal in the back frame buffer, before it is sent to the display. If rendering a frame takes too long and misses the next VSync signal, it must wait for the following VSync signal to be displayed.

Problems. The above VSync-driven rendering and displaying mechanism works well on a local system. However, in remote rendering,

[1] This limitation may be just due to cost or marketing strategy considerations, as consumer devices are mostly designed for decoding existing video streams rather than encoding new streams.

[2] As a result, B-frames can only be encoded and decoded when the next frame is available. Waiting for the next frame significantly increases the streaming latency and makes it infeasible to use B-frames for low latency VR systems. We do not use B-frames in our system.

[3] Most devices including smartphones are able to decode four or even more (e.g., eight) video streams simultaneously [11]. Thus, we do not need *decoder* multiplexing.



(a) Ideal case

(b) Long waiting time

(c) Missing a frame

Figure 5: Displaying two consecutive frames that are remotely rendered. (a) The ideal case where the frames are displayed immediately after they are ready. (b) The frames failed to meet their VSync signal and must wait for a long time before actually being displayed. (c) The frames become ready in the same VSync interval and thus frame *n* is dropped.

the frame displaying is driven by VSync signals of the HMD client but the frame rendering is driven by VSync signals of the rendering server. Due to the asynchronized frame rendering and displaying and the extra streaming latency, it may cause problems.

To illustrate the problems, we show the processing procedure of two consecutive frames n and $n+1$ in Figure 5. The rendering server starts to render frame n at time T_{render}^{n}. The frame is then encoded and transmitted to the client. The client decodes the frame and presents it to the frame buffer swap chain at time T_{ready}^{n}. We define the time interval between T_{render}^{n} and T_{ready}^{n}, i.e., $T_{ready}^{n} - T_{render}^{n}$, as the *generating time* of frame n. Ideally, the frame is ready just before VSync signal n at time T_{vsync}^{n}, so that it can be displayed immediately. This ideal case is shown in Figure 5(a). However, if frame n missed VSync signal n (i.e., $T_{ready}^{n} > T_{vsync}^{n}$), it must wait for VSync signal $n+1$ and thus the end-to-end latency is increased by $T_{vsync}^{n+1} - T_{ready}^{n}$. Such a long waiting time case is shown in Figure 5(b) [4]. Furthermore, if frame n missed the VSync signal n, and at the same time frame $n+1$ becomes ready before time T_{vsync}^{n+1} (i.e., $T_{ready}^{n+1} < T_{vsync}^{n+1}$), frame n will become useless and thus will be dropped. Instead, frame $n+1$ will be displayed upon VSync signal $n+1$. This case is called *frame missing* as shown in Figure 5(c) [5]. In this case, the time and resources used to render, encode, transmit and decode frame n are wasted.

To confirm that the two problems are real, we conduct an experiment. In this experiment, we cap the frame rate on the rendering server to 90 Hz. On the client, for each frame n, we record the time interval ΔT^{n} between the frame-ready time T_{ready}^{n} and the next VSync signal time after T_{ready}^{n}. We define such a time interval ΔT^{n} as the *waiting time* of frame n. Figure 6 plots the waiting time of more than 200 consecutive frames. It shows that the frame waiting time keeps drifting from 0 ms to 11.1 ms periodically. This is because that the rendering server is unaware of the VSync signals of the client and thus cannot synchronize its rendering with the frame displaying on the client. This phenomenon not only introduces additional latency as shown in Figure 5(b), but also causes frame missing when ΔT^{n} jumps in consecutive frames, which is shown in the red dotted circle in Figure 6. In the red dotted circle, for a frame n with a very large ΔT^{n} close to 11 ms, the ΔT^{n+1} of the next frame $n+1$ may immediately become very small close to 0 ms. In this case, the two frames n and $n+1$ are ready to display within the same VSync interval and thus frame n will not be displayed, which is the frame-missing case in Figure 5(c).

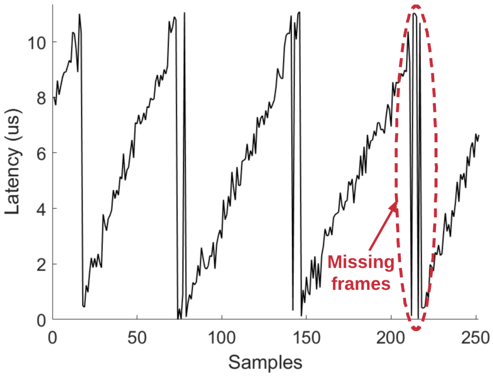


Figure 6: Time till next VSync signal.

Solution. To solve the problems, we propose to drive the frame rendering of the server using the VSync signals of the client. The key idea is adjusting the timing of rendering the next frame according to the feedback from the client on how the previous frame was displayed, how long its waiting time was, and how fast the HMD moved. Specifically, we use the following equations to decide when to start to render a new frame $n+1$:

$$T_{render}^{n+1} = T_{render}^{n} + \frac{1}{90}s + T_{shift} \tag{4}$$

$$T_{shift} = (T_{vsync}^{n} - T_{ready}^{n} - T_{conf} - T_{motion}) * cc \tag{5}$$

[4] This case may also happen in local systems without remote rendering. However, the extra streaming latency in remote rendering makes this case happen more frequently.
[5] This case is caused by remote rendering and will not happen in a local system unless VSync is disabled.

$$T_{motion} = k * \Delta\theta^n \tag{6}$$

In Equation 4, besides using a constant time interval of 1/90 seconds to maintain the frame rate of 90 Hz, we further introduce a time shift T_{shift} that dynamically modifies the start time of rendering frame $n+1$ (i.e., T^{n+1}_{render}). T_{shift} is decided by several factors. The first factor is the waiting time of frame n, $T^n_{vsync} - T^n_{ready}$. We intend to postpone T^{n+1}_{render} for time interval $T^n_{vsync} - T^n_{ready}$. Assume frames n and $n+1$ have the same generating time, i.e., $T^{n+1}_{ready} - T^{n+1}_{render} = T^n_{ready} - T^n_{render}$, the time of placing frame $n+1$ to the swap frame buffer chain T^{n+1}_{ready} exactly equals to T^{n+1}_{vsync}. This way, the waiting time of frame $n+1$ is minimized. However, if it takes a slightly longer time period to generate frame $n+1$, frame $n+1$ may miss its VSync signal $n+1$, resulting in a very long waiting time or even missing frame $n+1$. To mitigate this issue, we introduce T_{conf} to shift T^{n+1}_{ready} back a lit bit to tolerate the variance in generating frames.

We take a data-driven approach to decide a proper value for T_{conf}. Initially, we set T_{conf} to zero. We track the frame-generating time of the last 1,000 frames. We calculate the value that covers the variances of the frame-generating time of the last 1,000 frames with a confidence of 99% (a.k.a 99% confidential interval). We set the value of T_{conf} to the half of the 99% confidential interval. Over time, T_{conf} acts as an adaptive safeguard to handle the variance in generating consecutive frames.

Another factor we consider in T_{shift} is how fast the HMD moves. The intuition behind this consideration is as follows: when the HMD moves fast, the view of the VR game may change fast. As a result, the content of the next frame may have significant changes and thus its rendering time might be longer than that of its previous frame. To accommodate the large rendering cost of the next frame, we need to start to render it early to avoid missing VSync. We use T_{motion} for this purpose in Equation 5. However, as the frame rate is as high as 90 Hz, the absolute distance that the player may move along a direction in a frame time (i.e., 1/90 seconds) is pretty small and thus has limited impact on the content changes of the next frame in practice. But, the rotation of the HMD may significantly affect the content of the next frame due to the change of the viewing angle. Therefore, we only consider the viewing angle changes in our design. As shown in Equation 6, T_{motion} is determined by the change of viewing angle $\Delta\theta^n$ together with a constant scaling parameter k. We empirically set the value of k to 100.

Finally, we use a scaling parameter cc as a low-pass filter in calculating the value T_{shift}, as shown in Equation 4. We empirically set its value to 0.1.

With this *remote VSync driven rendering* approach, we try to ensure that the system can stay in the ideal case shown in Figure 5(a). We expect that most frames become ready to display just before the next VSync signal and thus have a very short waiting latency, and that very few frames are dropped. Furthermore, even we may postpone the rendering of a frame, we always use the latest possible pose of the HMD to render the frame. This is achievable because the sampling rate of HMD pose is as high as 1,000 Hz, one order of magnitude higher than the frame rate. As we display the postponed frame with the best possible VSync signal, we minimize the user-perceived latency and thus provide the best user experience. Indeed, we may even be able to do better than the tethered system. As we will show in Section 7, if the rendering time of a frame is very short, we may delay its rendering to reduce its waiting time. As we render it with a fresher HMD pose, doing so not only achieves a lower end-to-end latency, but also provide a better user experience, compared to the tethered system.

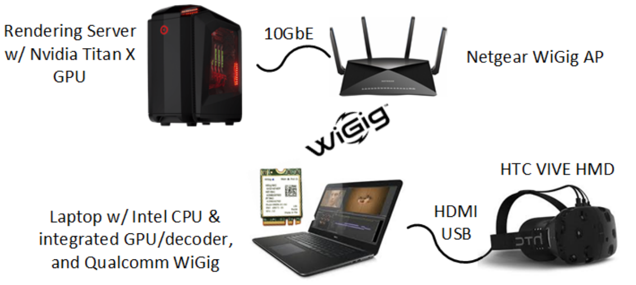

Rendering Server w/ Nvidia Titan X GPU
10GbE
Netgear WiGig AP
WiGig
HTC VIVE HMD
HDMI
USB
Laptop w/ Intel CPU & integrated GPU/decoder, and Qualcomm WiGig

Figure 7: Hardware setup.

6 IMPLEMENTATION

We implement our system on Windows for its rich supports on VR. We use Qualcomm WiGig solution to achieve 2.1 Gbps wireless transmission throughput. Our implementation is entirely based on commodity hardware and consists of around 7,000 lines of code.

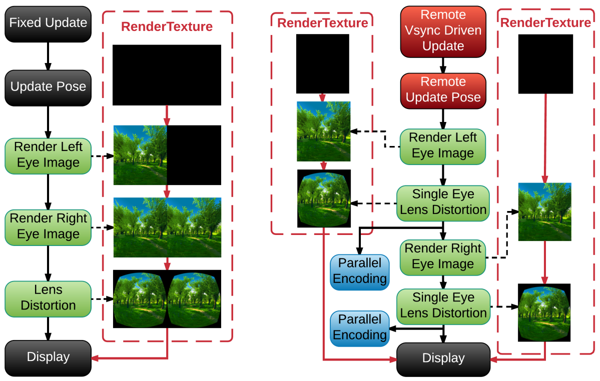
6.1 Hardware Setup

As shown in Figure 7, the rendering server is an Intel Core i7 based PC with a Nvidia TITAN X GPU. It has a Mellanox 10Gbps network interface card to connect to a Netgear Nighthawk X10 WiGig AP using a 10Gbps Ethernet cable. We use a ThinkPad X1 Yoga laptop to act as the client that connects to the Wigig AP through a Qualcomm QCA6320/QCA6310 WiGig module. The laptop equips an Intel i7-7600U CPU and an HD 620 Integrated GPU with H.264 hardware decoder ASIC included. The laptop connects to an HTC Vive HMD using HDMI and USB.

6.2 Software Implementation

We implement our proposed techniques based on the OpenVR SDK [18], the Unity game engine [19], and the Google VR SDK for Unity [20].

Remote Rendering VR Camera. The core of the server-side implementation is a VR camera that leverages Unity's rendering solution and Nvidia's Video Codec for low-latency remote rendering. This VR camera is modified based on the solution from the Google VR SDK [20]. Figure 8 shows the life cycle comparison between the normal VR camera used in the Google VR and our VR camera. The normal VR camera starts rendering each frame from a fixed update callback function, driven by the periodic VSync signals of the system. Then, the camera updates user's pose and rotates itself towards the correct direction in the 3D scene. After that, the camera renders the left eye image and the right eye image sequentially, then applies lens distortion on the whole frame. On the right side



(a) Normal VR Camera.

(b) Our VR Camera.

Figure 8: Life cycle comparison between our VR camera and normal VR camera.

of Figure 8(a), we show the changes of the RenderTexture in each step. A RenderTexture is a texture that can be rendered to by D3D or OpenGL. In the normal VR camera case, a screen-size RenderTexture is allocated when the VR camera is created. The camera renders left eye image and right eye image to each side of the render texture sequentially. Then, a dedicated vertex shader is applied to the whole texture to do the VR lens distortion [6].

Our VR camera works as shown in Figure 8(b). The fixed update function is changed to a remote VSync driven update function to optimize the rendering time based on the VSync signals on the HMD client. VSync time can be retrieved from the client side by calling the GetTimeSinceLastVsync() in the OpenVR API [18]. Similarly, the pose update function is changed to the remote pose update to get the pose information from the client. In this function, the server sends a request packet to the client through the wireless network. The client calls the GetDeviceToAbsoluteTrackingPose() function in the OpenVR API to get the current pose of both the HMD and controllers, and send them back to the server. Instead of using a single RenderTexture as the render target, we create two RenderTextures of half screen size for the image of each eye separately. We modify the distortion shader to work for only single eye image, and apply it on the RenderTexture immediately after rendering the image of one eye.

Parallel Encoding. Our parallel encoding module is developed as a Unity native plugin attached to the remote rendering VR camera. This module uses zero-copy between rendering and encoding in the GPU. We register the RenderTexture of each eye using the CUDA function cudaGraphicsD3D11RegisterResource() so that it can be accessed from CUDA. We directly pass the registered memory address to Nvidia's hardware video codec for encoding, and thus do not need any memory copy. As mentioned in Section 4, the system passes each RenderTexture to two separate encoders for parallel encoding, and creates total 4 encoding streams. The encoding operation is executed asynchronously without blocking the rendering of frames. To maintain a high image quality with low latency, we use the Two-Pass Rate Control setting with 400KB maximum frame size to encode each frame. With this setting, the encoded frame size is capped to 400KB.

Mobile VR	Latency (ms)	Frame Rate (Hz)	Visual Quality (SSIM)	Resolution (pixels)
Flashback [12]	25	60	0.932	1920x1080
Furion [11]	25	60	0.935	2560x1440
Ours	20	90	0.987	3840x2160

Table 1: Comparison between different mobile VR systems.

Parallel Decoding. We implement the parallel decoding module on the commercial Intel-based low power video codec, using the Intel Media SDK [21] that can be run on any 3rd generation (or newer) Intel Core processors. In this module, four decoding sessions are created to decode four encoded streams in parallel. A separate display thread is developed to keep querying the decoded frame blocks from the decoder sessions and display them on the corresponding positions on the HMD screen. This module also calls the OpenVR API to retrieve the pose and VSync data for sending to the rendering server.

7 EVALUATION

We evaluate the performance of our system in terms of end-to-end latency, the trade-off between visual quality and add-on streaming latency, frame missing rate, and resource usage on the client. We demonstrate that our system is able to achieve both low-latency and high-quality requirements of both the tethered HTC Vive VR and future VR at 4K resolution at 90Hz over a stable 60GHz wireless link. The result shows the system can support current 2160x1200 VR resolution with 16ms end-to-end latency and 4K resolution with 20ms latency. Furthermore, we show that our system misses very few frames in different VR scenes and uses only a small portion of CPU and GPU resources on the client.

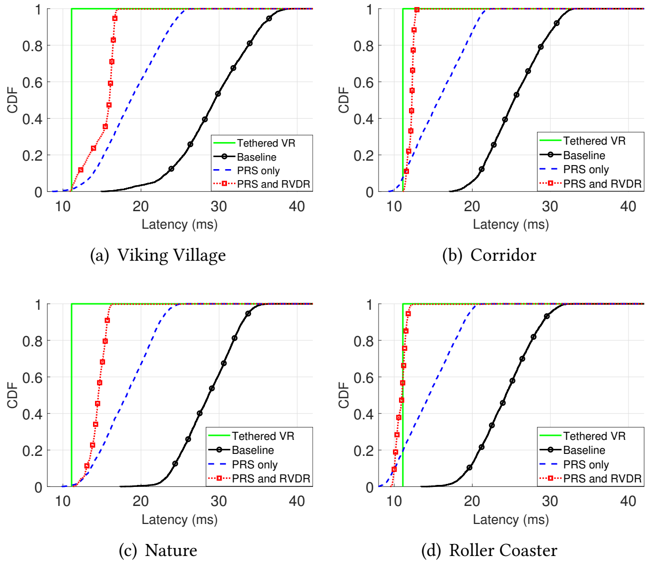
As shown in Table 1, our system outperforms previous untethered VR system [11, 12] in all four aspects: end-to-end latency, frame rate, visual quality, and resolutions. However, as we mentioned, we target to optimize the system issues arise when combining the whole rendering and streaming pipeline, thus is complementary to previous work. Performance may further improve when combining different approaches.

7.1 Experiment Setup

We use the hardware setup described in Section 6 to conduct experiments. Since our objective is not to improve 60GHz communication but provide an open platform to do so, we keep the wireless link stationary all the time [7]. We compare our system with the tethered HTC Vive VR system and a baseline untethered solution. As described in Section 4, the baseline solution is the typical video

[6] For illustration purpose, we set the original RenderTexture in Figure 8 as a black frame. In real systems, it should be the last frame when starting rendering the current frame.

[7] Mobility is an issue in existing 60GHz wireless products and there is active on-going research on addressing that issue [7, 8, 10].



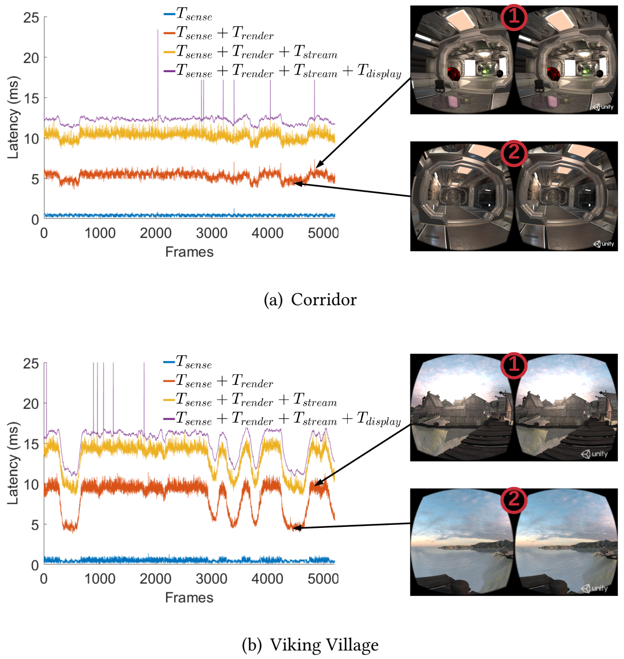
(a) Viking Village

(b) Corridor

(c) Nature

(d) Roller Coaster

Figure 9: CDF of end-to-end latency of 4 different approaches in 4 VR scenes.



(a) Corridor

(b) Viking Village

Figure 10: Raw latency traces of our system in running Corridor and Viking Village.

streaming approach that executes the rendering, encoding, transmission, and decoding sequentially, without parallel pipelines. We use four different VR scenes in our evaluation: *Viking Village* [22], *Nature* [23], *Corridor* [24], and *Roller Coaster* [25]. These four scenes are carefully selected to cover different kinds of VR applications. Viking Village and Nature are rendering intensive, requiring up to 11ms to render a frame even on our rendering server. Viking Village is a relatively static scene, while Nature has more than hundreds of dynamic objects (leaves, grasses, and shadows) that keep changing all the time. Corridor and Roller Coaster have relatively light rendering loads. Different from the other three scenes, Roller Coaster lets a player sit on a cart of a running coaster. Thus, the player's view keeps changing even if the player doesn't move at all.

7.2 End-to-end Latency

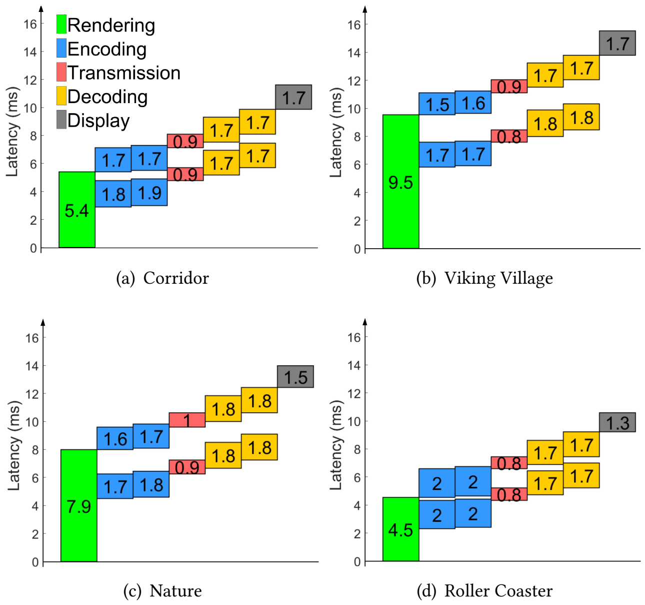
We first measure the end-to-end latency (T_{e2e}) in four cases: the tethered HTC Vive VR system (Tethered VR), the baseline untethered solution (Baseline), our solution with only the PRS technique (PRS only), and our solution with both PRS and RVDR techniques (PRS and RVDR). For repeatable experiments, we pre-logged a 1-min pose trace for each VR scene and replayed it for the experiments of the same scene in the four cases [8].

Figure 9 shows the CDFs of the measured results. In Tethered VR, T_{e2e} is always 1/90 seconds for the periodic VSync signals at 90Hz. In Baseline, T_{e2e} is very large due to the large extra cost of the sequential frame rendering, encoding, transmission, and decoding. The median value of T_{e2e} is more than 26ms and the maximum value is even larger than 38ms. Due to asynchronized VSync signals on the client and the server, the variance of T_{e2e} is also very large. With the PRS technique, we reduce T_{e2e} for more than 10ms, compared to Baseline. Combing the techniques of both PRS and RVDR, we not only reduce the average T_{e2e} to only 10-14ms depending on which scene is used, but also largely reduce the variance of T_{e2e}. For the scenes with light rendering loads, such as Corridor and Roller Coaster, our solution can achieve a comparable performance to Tethered VR. In Roller Coaster, we may even achieve a lower end-to-end latency (i.e., < 11ms) for many frames, because we render them with a fresher pose.

From Figure 9, it shows that T_{e2e} varies in different VR scenes. To figure out the reason, we plot the raw latency traces of two VR scenes (Corridor and Viking Village) in Figure 10. We breakdown the total end-to-end latency into four parts/tasks: T_{sense}, T_{render}, T_{stream}, and $T_{display}$. Each curve in Figure 10(a) and Figure 10(b) shows the total latency after each task is finished. For example, the curve at the bottom is T_{sense}, and the curve on the top is the sum of T_{sense}, T_{render}, T_{stream}, and $T_{display}$ (i.e., the total end-to-end latency T_{e2e}). Figure 10 shows that the rendering latency T_{render} is a critical part that affects the variance of the total end-to-end latency T_{e2e}. For Corridor, our system achieves a similar T_{e2e} compared to Tethered VR, because the rendering time $T_{rendering}$ is small (around 5ms). However, for rendering-intensive Viking Village, the rendering time $T_{rendering}$ is large (up to more than 10ms), resulting in a large T_{e2e}.

Furthermore, we observed that T_{e2e} has a small variance in Corridor, but a large one in Viking Village, fluctuating from 11ms to 15ms. This difference is also caused by rendering time. For illustration, we show two frames from each game in Figure 10. Each frame points to its rendering time on the corresponding rendering latency curve.

[8] We replay the logged traces on the client upon the requests from the server. Thus, the measured end-to-end latency includes the time cost of sensor data acquisition.



(a) Corridor

(b) Viking Village

(c) Nature

(d) Roller Coaster

Figure 11: Average latency (in ms) of each frame-processing step in the parallel pipeline.

In Corridor, the rendering time of different frames is relatively constant. In Viking Village, frame #2 (the user looked towards the water) has much less rendering load than frame #1, when the user faced the village. When the user moves, the rendering latency keeps fluctuating, and the total latency changes accordingly. However, as we apply the constant bitrate control setting in encoding each frame, the add-on streaming latency T_{stream} stays at a constant value no matter how the rendering latency changes.

Latency distribution in each step. Figure 11 shows the average latency of each step in frame rendering, encoding, transmission, decoding and displaying, and how the steps are overlapped. It shows that our PRS technique is able to build a very effective parallel pipeline to reduce the end-to-end latency. The average display time is only less than 1.7ms, demonstrating the effectiveness of our RVDR technique in reducing the display latency.

Frame rate. With the low add-on streaming latency in our system, we can actually enable a frame rate higher than 90Hz. Indeed, our system is able to achieve 150 frames per second in Corridor, demonstrating the capability of our system in supporting future VR at a higher frame rate, e.g., 120Hz.

7.3 Add-on Streaming Latency vs Visual Quality

The visual quality of encoded frames plays a critical role in providing a good user experience. Our system not only achieves low-latency remote rendering, but also keeps a visually lossless (visually identical) experience to users. It is well known that there's a trade-off between the visual quality and the streaming latency. In our system, visual quality is controlled by the bitrate in the Rate Control settings of encoding. To quantify the visual quality of an encoded frame, we use the widely used Structural Similarity (SSIM) to determine how similar the encoded frame is to the original frame. Table 2 shows the average SSIM score of the four VR scenes in different bitrate settings (each results in a different encoded frame size). From a previous study [26], a visually lossless encoded visual requires the SSIM score to be larger than 0.98. In Table 2, it shows that the encoded frame size of 400KB or larger can achieve an SSIM score of more than 0.98 for all the four VR scenes.

Frame Size	Nature	Corridor	Viking	Roller
800KB	0.9914	0.9936	0.9927	0.9965
400KB	0.9831	0.9887	0.9865	0.9913
200KB	0.9707	0.9780	0.9763	0.9838
100KB	0.9560	0.9609	0.9643	0.9718
60KB	0.9411	0.9439	0.9478	0.9605

Table 2: Encoded frame size and visual quality measurement (SSIM score).

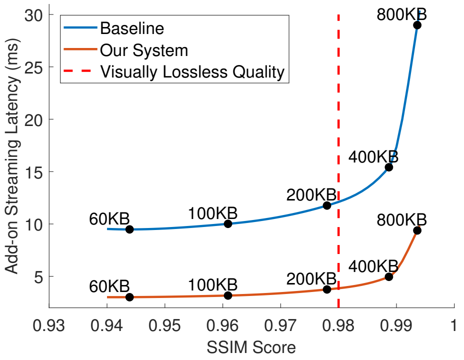


Figure 12: Add-on streaming latency vs visual quality for Corridor.

To further build a relationship between add-on streaming latency and visual quality, we calculate the add-on streaming latency in different encoded frame sizes. Figure 12 shows how T_{stream} changes with the image SSIM score in Corridor, with the comparison of the Baseline approach and our approach. We also draw the visually lossless quality cutting curve as a red dash line. It shows the system requires an encoded frame size of more than 200KB to achieve visually lossless quality. The add-on streaming latency requirement to achieve this quality in our system is only around 4ms which is much smaller than the one of the Baseline approach.

7.4 Frame Missing Rate

In Figure 10, each pin-shape peak on the top purple curve represents a missing frame. To quantify the effectiveness of the RVDR technique in reducing frame missing, we calculate the average frame-missing rate in the four VR apps with and without RVDR enabled. Figure 13 shows the results. We can see that RVDR is able to significantly reduce the frame-missing rate. Without RVDR, the average frame-missing rate is from 5.3% to 14.3%. With RVDR, it is reduced to only 0.1% - 0.2%.

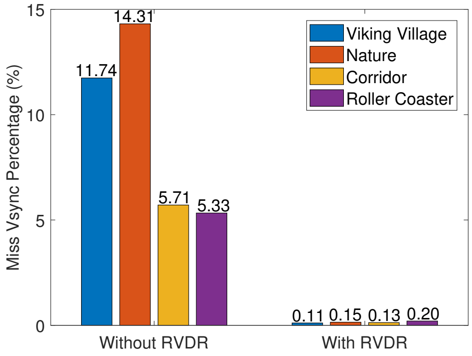


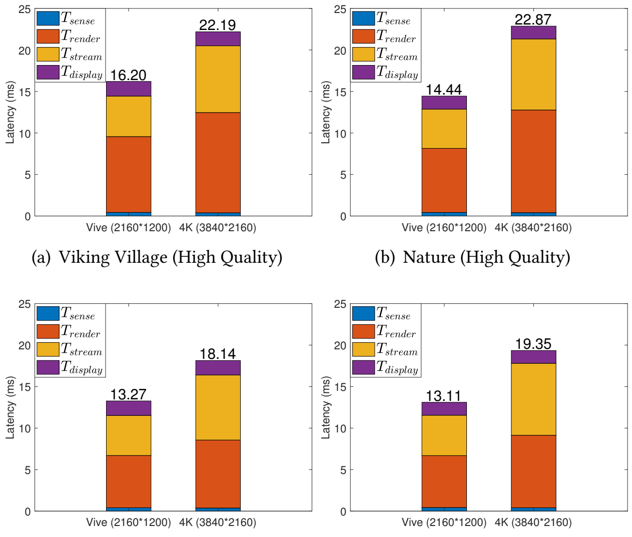
Figure 13: Frame missing rate with and without RVDR in the 4 VR apps.

7.5 Resource Usage on Client

We also measure the resource usage on the client. Without our system, Windows 10 uses 3% to 5% CPU. With our system playing a VR game, the CPU usage is around 36% mainly for handling the network packets. Decoding frames use only 29% of the video-decoding capability of the GPU (not the whole GPU capability such as rendering 3D scenes). In building a future untethered HMD using our techniques, such resource usage may be further significantly reduced by a more integrated hardware design with embedded software. We further discuss the power consumption of the system in Section 9.

7.6 4K Resolution Support

To expand our system towards future VR systems, we further measure whether our system can deliver 4K resolution (3840x2160) VR frames with less than 20ms end-to-end latency. To do it, we conduct experiments on the 4 VR scenes with Vive (2160x1200) and 4K resolutions. We force the rendering engine to render 4K frames with the same graphics quality settings as the Vive case. To achieve visually lossless encoding of 4K, we enlarge the maximum encoded frame size to 500KB. With the 4K resolution, both T_{render} and T_{stream} are larger than those of Vive resolution in all four VR scenes. The average T_{stream} to stream a 4K resolution frame is 8.3ms. The average T_{sense} and $T_{display}$ are 0.4ms and 1.7ms, respectively, not affected by the resolution change. For Corridor and Roller Coaster, our system can still keep the T_{e2e} within 20ms, but the T_{e2e} of Viking Village and Nature exceed 20ms. Figure 14(a) and 14(b) show the latency breakdown of the 2 scenes. The rendering latency T_{render} is the bottleneck in 4K. This is because the GPU we used is not 4K VR ready, and thus both scenes require more than 11ms to render a 4K frame, which cannot meet the frame rate of 90Hz. While this issue may be solved by using a 4K VR ready GPU, we want to study whether our system has the capability to deliver 4K frames with 90Hz if such a GPU is available. Therefore, we reduce the rendering quality of Viking Village and Nature to squeeze the rendering latency to less than 11ms. As shown in Figure 14(c) and 14(d), by doing so, our system is able to meet the requirement of 20ms end-to-end latency. With average $T_{sense} = 0.4ms$, $T_{display} = 1.7ms$, and $T_{stream} = 8.3ms$, our system is able to support 4K resolution if the rendering time is within 9.6ms.



(a) Viking Village (High Quality)

(b) Nature (High Quality)

(c) Viking Village (Low Quality)

(d) Nature (Low Quality)

Figure 14: Latency breakdown in 4K resolution.

8 RELATED WORK

Cutting the Cord. Cutting the cord of high-quality VR systems has attracted strong interest from both industry and academia. TPCAST [13] and Display Link XR [14] provide a wireless adapter for HTC Vive. They both take a direct “cable-replacement" solution that compresses the display data, transmits over wireless and decompresses on HMD. To our knowledge, they compress whole frames after the frames are fully rendered, and thus it is hard to explore the opportunities to pipeline rendering/streaming and fine-tune rendering/VSync timing. Moreover, their solutions are implemented in ASICs instead of commodity devices. Therefore, they are hardly used by the research community to explore advanced system optimization. Zhong *et al.* [15] explored how to cut the cord using commodity devices and measured the performance of different compression solutions on CPUs and GPUs. Their measurement results provide valuable guidance and are motivating to us.

High-Quality VR on Mobile. Mobile devices are another popular VR platform. Google Daydream [6], Google Cardboard [27] and Samsung Gear VR [5] are examples of this type. As we have discussed, it is very difficult to achieve high-quality VR on mobile due to its limited computing power. Some research works have tried to attack this problem. For example, Flashback [12] performs expensive rendering in advance and caches rendered frames in mobile devices. Doing so provides high-quality VR experience for scenarios that can be pre-computed. Furion [11] enables more scenarios by offloading costly background rendering to a server and only performs lightweight foreground rendering on a mobile device. Such collaborative rendering reduces overall rendering time, which is complementary to our design and can be incorporated into our system to further reduce the latency. Similarly, mobile offloading techniques, e.g., [26, 28–32] also could be borrowed.

Wireless Performance. Performance of wireless link is critical to wireless VR experience. There are a lot of ongoing research on this issue, especially on mobility and blockage handling for 60GHz/mmWave. For example, MoVR [7] specially designs a mmWave communication system for wireless VR. Agile-Link [8] provides fast beam tracking mechanism. MUST [10] redirects to Wi-Fi immediately upon blockage. Pia [9] switches to different access point proactively. These studies are complementary to our system.

Video Streaming. Our work is also related to video streaming techniques. Nvidia's video codec [33] and Intel's Quick Sync [34] provide fully hardware-accelerated video decoding/encoding capability. Most of these techniques enable *slice-mode* video streaming, which cuts the whole frame into pieces and streams separately. We borrow this idea and combine it into the rendering pipeline to enable parallel rendering and streaming using multiple hardware encoders. 360-degree video streaming [35–40] pre-caches the panoramic view and allows users to freely control their viewing direction during video playback. Multi-view video coding [41–44] enables efficient images encoding from multi-viewpoint using both the temporal and spatial reference frames. Video streaming is not extremely latency sensitive as interactive VR unless the streaming is real-time (broadcasting). We may use some techniques in video streaming particularly 360-degree streaming to our system.

9 DISCUSSION

In this section, we discuss the following three issues: (1) advantages and the generality of our system, (2) power consumption, and (3) link outage limitations that were out of the scope of this project.

Generality. Our system is a software solution that can be extended to different hardware and operating systems. The server-side rendering module is developed using Unity, which is known to be compatible with D3D and OpenGL. The encoding module on the server side and the decoding module on the client side can be implemented using various hardware codec APIs, such as Nvidia Video Codec [33], Intel Quick Sync [34], Android MediaCodec [45], etc. Compared to previous approaches that still require additional rendering operations on the client side [11, 12, 26, 28], our solution only requires a hardware video codec on the client side, allowing it to work with very thin clients. Our system optimizations of the entire VR offloading pipeline can also be combined with previous techniques (e.g. pre-rendering background scenes [11], robust 60 GHz network [7]) to further improve the performance of the VR offloading task.

Power Consumption. To truly allow cutting the cord of an existing high-quality VR system, it is also important to consider the power consumption on the client side. Since we use the hardware codec from a laptop in our prototype, it is hard to decouple the codec power consumption from the overall laptop consumption. We therefore estimate power consumption based on reported power consumption data from the three main components (VR headset, H.264 hardware decoder, and WiGig wireless adapter) in our implementation by referring to previous measurement results [46–48]. The power consumption of an HTC VIVE when running normally is 5.9W [46], and the 802.11ad's power consumption is around 2.3W [47]. We also estimate the power draw of the H.264 decoder in our 4-way parallel decoding scenario based on a prior measurement result of an H.264 video decoder [48]. As shown in Table 3, the total power consumption estimate for these key components is 12W, which shows that such a system could be powered from a smartphone-sized Lithium-ion battery for about 3 hours. Note that this power consumption is estimated in a conservative way. The real consumption may be much lower with customized hardware design.

System component	VR headset	H.264 decoder	WiGig
Power (W)	5.9	4.8	2.3

Table 3: Power consumption of three main components in the system.

Limitations. This project has not addressed the link outage problem of 60GHz networks, which will likely require orthogonal solutions. To effectively evaluate the performance gain of our solution without the measurements being affected by random movement and link outages, we kept the 60GHz antennas stationary in our experiment to maintain a stable wireless link between the server and the client. It is well known that mobility is still an issue in existing 60GHz wireless products and there is active on-going research to address this [7, 8, 10]. We expect that this project provides a platform that enables such research with realistic end-to-end applications and that ultimately this research will lead to solutions that can be combined with the techniques presented here. We further note that our system can also operate over a Wi-Fi network, which is less susceptible to obstructions, albeit with sacrificing image resolution.

10 CONCLUSION

In this paper, we design an untethered VR system that is able to achieve both low-latency and high-quality requirements over a wireless link. The system employs a Parallel Rendering and Streaming mechanism to reduce the add-on streaming latency, by pipelining the rendering, encoding, transmission and decoding procedures. We also identify the impact of VSync signals on display latency, and introduce a Remote VSync Driven Rendering technique to minimize the display latency. Furthermore, we implement an end-to-end remote rendering platform on commodity hardware over a 60GHz wireless network. The result shows that the system can support 4K resolution frames within an end-to-end latency of 20ms. We plan to release the system as an open platform to facilitate VR research, such as advanced rendering technologies and fast beam alignment algorithms for 60Ghz wireless communication.

ACKNOWLEDGEMENTS

We sincerely thank our shepherd Iqbal Mohomed and anonymous reviewers for their valuable comments. Part of this work was done when Luyang Liu, Ruiguang Zhong, and Jiansong Zhang were with Microsoft Research. This material is based in part upon work supported by the National Science Foundation under Grant Nos. CNS-1329939.

REFERENCES

[1] Virtual Reality and Augmented Reality Device Sales to Hit 99 Million Devices in 2021. http://www.capacitymedia.com/Article/3755961/VR-and-AR-device-shipments-to-hit-99m-by-2021.

[2] The reality of VR/AR growth. https://techcrunch.com/2017/01/11/the-reality-of-vrar-growth/.

[3] HTC Vive VR. https://www.vive.com/.

[4] Oculus Rift VR. https://www.oculus.com/.

[5] Samsung Gear VR. http://www.samsung.com/global/galaxy/gear-vr.

[6] Google Daydream. https://vr.google.com/.

[7] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling high-quality untethered virtual reality. In *NSDI*, pages 531–544, 2017.

[8] Omid Abari, Haitham Hassanieh, Michael Rodriguez, and Dina Katabi. Millimeter wave communications: From point-to-point links to agile network connections. In *HotNets*, pages 169–175, 2016.

[9] Teng Wei and Xinyu Zhang. Pose information assisted 60 ghz networks: Towards seamless coverage and mobility support. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 42–55, New York, NY, USA, 2017. ACM.

[10] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. Wifi-assisted 60 ghz wireless networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 28–41, New York, NY, USA, 2017. ACM.

[11] 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 *Proceedings of the 23rd International Conference on Mobile Computing and Networking (MobiCom'17). ACM, Snowbird, Utah, USA*, 2017.

[12] Kevin Boos, David Chu, and Eduardo 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*, pages 291–304. ACM, 2016.

[13] TPCAST Wireless Adapter for VIVE. https://www.tpcastvr.com/product.

[14] DisplayLink XR: Wireless VR Connection. http://www.displaylink.com/vr.

[15] Ruiguang Zhong, Manni Wang, Zijian Chen, Luyang Liu, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Thomas Moscibroda. On building a programmable wireless high-quality virtual reality system using commodity hardware. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 7. ACM, 2017.

[16] Nvidia Video Encode and Decode GPU Support Matrix. https://developer.nvidia.com/video-encode-decode-gpu-support-matrix.

[17] Nvidia's VR ready GPUs. https://www.geforce.com/hardware/technology/vr/supported-gpus.

[18] OpenVR SDK. https://github.com/ValveSoftware/openvr.

[19] Unity game engine. https://unity3d.com/.

[20] Google VR SDK for Unity. https://github.com/googlevr/gvr-unity-sdk.

[21] Intel Media SDK. https://software.intel.com/en-us/media-sdk.

[22] Viking Village. https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140.

[23] Nature. https://assetstore.unity.com/packages/3d/environments/nature-starter-kit-2-52977.

[24] Corridor. https://assetstore.unity.com/packages/essentials/tutorial-projects/corridor-lighting-example-33630.

[25] Roller Coaster. https://assetstore.unity.com/packages/3d/props/exterior/animated-steel-coaster-plus-90147.

[26] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135. ACM, 2015.

[27] Google Cardboard. https://vr.google.com/cardboard/.

[28] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM, 2015.

[29] John R Lange, Peter A Dinda, and Samuel Rossoff. Experiences with client-based speculative remote display. In *USENIX Annual Technical Conference*, pages 419–432, 2008.

[30] Benjamin Wester, Peter M Chen, and Jason Flinn. Operating system support for application-specific speculation. In *Proceedings of the sixth conference on Computer systems*, pages 229–242. ACM, 2011.

[31] Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. Proxy-guided image-based rendering for mobile devices. In *Computer Graphics Forum*, volume 35, pages 353–362. Wiley Online Library, 2016.

[32] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. Towards efficient edge cloud augmentation for virtual reality mmogs. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 8:1–8:14, New York, NY, USA, 2017. ACM.

[33] Nvidia Video Codec. https://developer.nvidia.com/nvidia-video-codec-sdk.

[34] Intel Quick Sync. https://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html.

[35] Feng Qian, Lusheng Ji, Bo Han, and Vijay Gopalakrishnan. Optimizing 360 video delivery over cellular networks. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 1–6. ACM, 2016.

[36] Evgeny Kuzyakov and David Pio. Next-generation video encoding techniques for 360 video and vr.(2016). *Online: https://code. facebook.com/posts/1126354007399553/nextgeneration-video-encoding-techniques-for-360-video-and-vr*, 2016.

[37] Matthias Berning, Takuro Yonezawa, Till Riedel, Jin Nakazawa, Michael Beigl, and Hide Tokuda. parnorama: 360 degree interactive video for augmented reality prototyping. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 1471–1474. ACM, 2013.

[38] Mark Kressin. Method and apparatus for video conferencing with audio redirection within a 360 degree view, August 16 2002. US Patent App. 10/223,021.

[39] Amy Pavel, Björn Hartmann, and Maneesh Agrawala. Shot orientation controls for interactive cinematography with 360 video. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 289–297. ACM, 2017.

[40] Ching-Ling Fan, Jean Lee, Wen-Chih Lo, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Fixation prediction for 360 video streaming to head-mounted displays. 2017.

[41] Karsten Müller, Heiko Schwarz, Detlev Marpe, Christian Bartnik, Sebastian Bosse, Heribert Brust, Tobias Hinz, Haricharan Lakshman, Philipp Merkle, Franz Hunn Rhee, et al. 3d high-efficiency video coding for multi-view video and depth data. *IEEE Transactions on Image Processing*, 22(9):3366–3378, 2013.

[42] Ying Chen, Ye-Kui Wang, Kemal Ugur, Miska M Hannuksela, Jani Lainema, and Moncef Gabbouj. The emerging mvc standard for 3d video services. *EURASIP Journal on Applied Signal Processing*, 2009:8, 2009.

[43] Shinya Shimizu, Masaki Kitahara, Hideaki Kimata, Kazuto Kamikura, and Yoshiyuki Yashima. View scalable multiview video coding using 3-d warping with depth map. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(11):1485–1495, 2007.

[44] Anthony Vetro, Thomas Wiegand, and Gary J Sullivan. Overview of the stereo and multiview video coding extensions of the h. 264/mpeg-4 avc standard. *Proceedings of the IEEE*, 99(4):626–642, 2011.

[45] Android MediaCodec. https://developer.android.com/reference/android/media/MediaCodec.html.

[46] Vive Power Draw Test Results. https://www.reddit.com/r/Vive/comments/51ir1h/vive_power_draw_test_results/.

[47] Swetank Kumar Saha, Tariq Siddiqui, Dimitrios Koutsonikolas, Adrian Loch, Joerg Widmer, and Ramalingam Sridhar. A detailed look into power consumption of commodity 60 ghz devices. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2017 IEEE 18th International Symposium on*, pages 1–10. IEEE, 2017.

[48] H.264 HD Video Decoder Power Consumption. https://www.soctechnologies.com/ip-cores/ip-core-h264-decoder/.