Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices

Zeqi Lai, Y. Charlie Hu, *Fellow, IEEE*, Yong Cui, *Member, IEEE*,
Linhui Sun, Ningwei Dai, and Hung-Sheng Lee

Abstract—Despite the growing market penetration, today's high-end virtual reality (VR) systems remain tethered, which not only limits users' VR experience but also creates a safety hazard. In this paper, we perform a systematic design study of the "elephant in the room" facing the VR industry – is it feasible to enable high-quality VR apps on untethered mobile devices such as smartphones? Our quantitative, performance-driven design study makes two contributions. First, we show that the QoE achievable for high-quality VR applications on today's mobile hardware and wireless networks via local rendering or offloading is about 10X away from the acceptable QoE, yet waiting for future mobile hardware or next-generation wireless networks (e.g., 5G) is unlikely to help, because of power limitation and the higher CPU utilization needed for processing packets under higher data rate. Second, we present Furion, a VR framework that enables high-quality, immersive mobile VR on today's mobile devices and wireless networks. Furion exploits a key insight about the VR workload that foreground interactions and background environment have contrasting predictability and rendering workload, and employs a split renderer architecture running on both the phone and the server. Supplemented with video compression, use of panoramic frames, parallel decoding on multiple cores on the phone, and view-based bitrate adaptation we demonstrate Furion can support high-quality VR apps on today's smartphones over WiFi, with under 14 ms latency and 60 FPS (the phone display refresh rate).

Index Terms—Virtual reality, mobile devices, untethered, measurement

1 Introduction

The consumer virtual reality (VR) revolution started around 2012 when Palmer Luckey launched the now legendary Kickstarter campaign for Oculus Rift, which was acquired by Facebook in 2014. Since then, VR systems have been gaining growing market penetration, and are estimated to generate $30 billion annual revenue by 2020 [1].

Despite the growing market penetration, today's high-end VR systems such as Oculus Rift [2] and HTC Vive [3][1] that offer high quality and accurate positional tracking and handheld controllers remain tethered, where high-quality frames are rendered on a powerful server and streamed to the headset via a multi-Gbps HDMI cable. The cable not only limits users' mobility and hence VR experience but also creates a safety hazard as the user may trip and fall.

In this work, we perform a systematic design study to tackle this "elephant in the room" facing the VR industry – *is it feasible to enable high-quality VR apps on untethered mobile devices such as smartphones?*

We carry out our study in three steps. First, we profile the performance of two extreme design points on the latest Pixel phone which was dubbed by Google as "VR ready" [4], local rendering which performs rendering entirely on the phone, and remote rendering which renders frames entirely on a server and streams them over the best wireless network available on today's commodity smartphones, 802.11ac.

Our profiling study shows that the QoE achievable for high-quality VR apps on today's high-end mobile hardware (Pixel) and wireless networks (802.11ac) is about 10X away from the acceptable QoE. In particular, rendering a frame in a high-quality VR app on Pixel takes about 63-111 ms and exhausts the phone CPU and GPU with utilization at about 60 and 100 percent, while on-demand streaming a frame rendered at the server over 802.11ac takes about 230 ms.

Previous observation and reasoning [5] have suggested that waiting for tomorrow's mobile hardware and next-generation wireless networks is unlikely to directly help, and the likely path to support untethered high-quality VR, i.e., meeting its QoE requirements, is via software innovation. As the second step of our study, we develop such a software solution, called Furion. Furion is a VR framework that enables high-quality, immersive mobile VR on today's mobile device and wireless networks. The first thing Furion exploits is the classic technique for trading off computation time for reduced network transfer time: compression. However, we show directly applying the video compression scheme H.264 to compress the on-demand rendered frames on the server requires 32 ms to get the frame and 16 ms to decode it on the phone, for a total of 48 ms end-to-end

1. Including both HMDs and the software platform.

- *Z. Lai, Y. Cui, L. Sun, N. Dai, and H. Lee are with Tsinghua University, Beijing 100084, China. E-mail: {uestclzq, lh.sunlinh, lemondnw}@gmail.com, cuiyong@tsinghua.edu.cn, , lihs17@mails.tsinghua.edu.cn.*
- *Y. C. Hu is with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907. E-mail: ychu@purdue.edu.*

Manuscript received 24 Apr. 2018; revised 23 Feb. 2019; accepted 9 Apr. 2019. Date of publication 26 Apr. 2019; date of current version 3 June 2020.
(Corresponding author: Yong Cui)
Digital Object Identifier no. 10.1109/TMC.2019.2913364

1536-1233 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

delay, which is about 3X beyond the 16 ms per-frame interval on a 60 FPS display.

To bridge this 3X gap, the FURION design exploits several key insights about the VR workload: (1) a frame in high-quality VR apps consists of foreground interactions and background environment, (2) background environment is fairly predictable as it is incrementally updated as the user moves around in the virtual world, but has a much heavier rendering load due to rich details and complex textures, and hence is ideally suited for pre-rendering on and pre-fetching from the server, (3) foreground interactions are much lighter weight but unpredictable and hence are suited for rendering on the phone.

Motivated by the above insight, FURION employs a *cooperative renderer* architecture that renders less predictable, lighter-load interactions on the phone to *avoid network latency delay* while prefetching pre-rendered environment frames viewed from adjacent locations in the virtual world during the time it takes the user to walk to the adjacent location to *hide network transfer delay*.

While the cooperative renderer architecture is necessary to reduce end-to-end frame rendering time to under 16 ms, making it to work requires solving a few remaining engineering challenges: (1) At a location in the virtual world, the user may still change orientation abruptly. How to render a new environment corresponding to any orientation without fetching new frames from the server on-demand? (2) How to prefetch the needed pre-rendered environment frames from the server just in time to be used for the next frame?

To tackle the first challenge, we prefetch pre-rendered panoramic frame of the environment for adjacent locations in the virtual world, which can be used to render an environment frame of any orientation by cropping. To tackle the second challenge, we use video compression to drastically cut down the panoramic frame size, and we further cut each panoramic frame into multiple segments to enable parallel decoding on multiple cores of the phone.

Moreover, in the third step of our study, to further extend the flexibility and applicability of FURION framework, we propose a *flexible background adaptation* scheme to enable high-quality and low-latency VR under various network conditions (e.g., the server is deployed on a remote server) by dynamically adjusting bitrate according to current field of view (FoV). Intuitively, the key idea used in our scheme can be summarized as prefetching more nearby reachable frames during the movement to tolerate higher network latency and properly reducing the bitrate of a frame according to FoV to match the available bandwidth in real time.

We have implemented the complete FURION platform on top of Unity [6] and Google Daydream [7]. We further show the FURION platform is easy to use—VR apps developed with Unity can be easily augmented to run on top of FURION.

Finally, we evaluate FURION using three open-source high-quality VR apps from Unity ported to Google Daydream, Viking Village [8], Corridor [9] and Nature [10]. We show FURION can support high-quality VR apps on today's smartphone (e.g., PIXEL) over WiFi (802.11ac), by providing under 14 ms latency and 60 FPS (the phone display refresh rate), under diverse user interactions including controller, rotation, and movement in the virtual world, for the three high-quality VR apps. We also show that FURION scales well with app features, in terms of the number of both multiple dynamic objects in the environment and multiple foreground interactions which enhance the immersive experience of VR apps. We further show that FURION supports the above QoE while incurring acceptable resource usage (under 37 GPU usage and 65 percent CPU utilization), which allows it to sustain long running of VR apps without being restricted by temperature control and frequency capping. Finally, we evaluate the performance of FURION with flexible background adaptation under various network conditions. Since FURION dynamically adjusts the prefetching area and selects proper bitrate according to real-time network condition and user's FoV, FURION significantly outperforms other state-of-the-art schemes in terms of visual quality and responsiveness. In summary, our evaluation shows that FURION's cooperative renderer design presents a viable solution to support high-quality VR apps on today's mobile hardware and wireless networks.

```
while(true) //VR loop
{       //Stage 1: sampling
    SamplePose();//read pose from sensor
    SampleController();//read input from controller
    //Stage 2: updating
    Update();//update app state/logic
    //Stage 3: rendering
    RenderNewFrame(){
        drawForeground();//draw foreground in the buffer
        drawBackgound();//draw background in the buffer
    }
    //Stage 4: displaying
    DisplayNewFrame();
}
```

Fig. 1. The main render loop of a typical VR system.

2 MOTIVATION

2.1 VR Background

A typical VR system consists of three key components: a VR *headset*, a *controller* and a *renderer*. The headset serves two purposes: tracking user *pose* (including both 3D position and 3D orientation) and presenting the VR content to the headset display for user viewing. The controller integrates several physical buttons, touchpads and sensors to receive user interactions other than user pose. Additionally, some VR systems that contain social elements and support multiple online players can also receive remote interactions from other players via the Internet. The renderer then renders new frames according to new pose and interactions, which are updated on the head-mounted display.

Like the majority of game apps, most VR apps are built upon a render loop, which repetitively executes the steps as shown in Fig. 1: (1) sampling pose/interaction from the headset/controller; (2) updating application-specific state or logic; (3) rendering a new frame; and (4) displaying the new frame on the screen. Typically the running time of one iteration is defined as frame latency, which is dominated by the rendering time. Frame latency is a key metric for evaluating the responsiveness of the VR system since it reflects the delay between a user's input and the corresponding output.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

2.2 QoE Requirements of VR Apps

VR apps impose a heavy computation workload on the system since they have to track user pose and interactively render high-quality graphics in real time. Any performance degradation can result in user discomfort or motion sickness due to the near-eye display characteristics of VR systems [11]. Specifically, to support acceptable user experience, which is dictated by the human biology, modern VR systems have to meet three critical performance and feature requirements (i.e., the QoE requirements):

1) *Responsiveness.* The motion-to-photon latency should be low enough, e.g., under 10-25 ms [12], in order to sustain good user experience.
2) *High-quality visual effects.* The VR system should not only render photo-realistic scenes in presenting the immersive virtual environment for users, but also ensure seamless playbacks by offering high frame rates (e.g., 60 FPS is a minimum target [11]).
3) *Mobility.* A tethered VR headset presents a potential tripping hazard for the user. The VR system or at least the headset component should be untethered in order to provide the ultimate VR experience [13], [14].

2.3 High-Quality Mobile VR is Delay Limited

Accomplishing the first two QoE requirements simultaneously for high-quality VR apps, however, has remained unattainable today on mobile systems due to the limited capability of mobile hardware and wireless technologies. As a result, all current high-end VR systems (e.g., Oculus-Rift [2], HTC Vive [3]) are *tethered*, providing good visual quality and responsiveness at the cost of mobility. In particular, these systems execute the heavy workload on powerful servers, and stream rendered frames to the headsets over high bandwidth cables.

Cutting the cord of high-quality VR systems is challenging because of the mismatch between their strict QoE requirements and mobile computation/network capability. To understand the gap between the two QoE requirements of high-quality VR apps (latency and FPS) and what today's mobile hardware and networking capability can deliver, we experimentally measure the achievable QoE under two straight-forward approaches to support mobile VR: local rendering and remote rendering.

2.3.1 Approach 1: Local Rendering

The straight-forward approach to enable untethered VR systems is to perform rendering entirely on the mobile device, or local rendering for short. In fact, local rendering has been recently made available on commercial mobile VR systems such as Google Daydream [7] and Samsung Gear VR [15] to interactively render VR contents via the CPU/GPU on the smartphone. However, today's mobile VR systems can only support VR apps with low graphical quality which often breaks the illusion of immersion [11], [16], as they cannot satisfy the minimum QoE in running high-quality VR apps due to the constrained computation resource.

To quantify the resource usage and achieved QoE of popular VR apps on today's high-end mobile device, we experimented with seven popular VR apps on Google Daydream, as listed in Table 1: PolyRunner [17], Lego [18], vTime [19] and Overlord [20] are popular but low-quality Android VR apps available in Google Play which all have controller interactions, and Viking Village [8], Corridor [9] and Nature [10] are high-quality virtual-world Unity apps with more complex, realistic and immersive environments as shown in Fig. 2. Because Viking, Corridor and Nature are three high-quality Unity apps originally designed for high-end PCs, we used Google Daydream SDK to port them to run on Android and added support for rendering virtual reality as well as controller interactions.

TABLE 1
QoE and CPU/GPU Load When Running Different VR Apps on Pixel

App Name	BRISQUE [21] Score	FPS	CPU Load	GPU Load
Low-quality VR apps				
PolyRunner [17]	89.19	60	36.2%	68.9%
Lego [18]	59.31	52	39.3%	74.4%
vTime [19]	54.48	47	41.7%	70.3%
Overlord [20]	50.10	41	44.8%	84.9%
High-quality VR apps				
Viking [8]	24.37	11	55.8%	99.9%
Corridor [9]	26.67	9	55.5%	99.8%
Nature [10]	26.67	16	59.9%	99.8%

A lower BRISQUE value of an app indicates better visual quality.

We ran the seven VR apps on a Pixel smartphone running Android 7.1. We used FPS and the BRISQUE value [21] to quantify the QoE, where the BRISQUE value is a well-known non-reference image quality access (IQA) metric. A lower BRISQUE value indicates better image quality and higher visual complexity for an app.[2]

The results in Table 1 show that (1) even the first four, low-quality VR apps incur up to 85 percent GPU utilization, while driving the CPU at 40 percent utilization[3] on average, and yet could only achieve 41 to 60 FPS; (2) the three high-quality VR apps, Viking, Corridor and Nature, exhausted both the GPU (100 percent utilization) and CPU (55-59 percent utilization), yet could achieve only 9-16 FPS, which makes the apps not usable. These results suggest that the VR rendering workload is simply too heavy for today's high-end mobile systems such as Pixel and as a result the VR apps suffer a rendering delay of 63-111 ms which is far longer than the required 16 ms per-frame rendering interval.

To summarize, under local rendering, the time to render a frame (roughly the inverse of each app's FPS which is measured by Google Daydream SDK [7]) is:

$$T_{phone_render} = 63 \sim 111 \text{ ms}. \quad (1)$$

2.3.2 Approach 2: Remote Rendering

The overwhelming CPU/GPU demand of local rendering on mobile devices motivates the alternative approach to support untethered VR of offloading the high rendering

2. There is no well-defined threshold BRISQUE value for defining acceptable image qualities.

3. Throughout the paper, phone CPU utilization refers to the average utilization of all cores.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.



(a) PolyRunner (b) Viking Village (c) Corridor (d) Nature

Fig. 2. A subset of VR apps used in our study: PolyRunner is low-quality, and the rest are high-quality.

load to a powerful server and streaming the newly rendered frames to the headset via the wireless network, or remote rendering for short. In essence, this approach trades off computation workload for network workload. However, such a tradeoff essentially turns the overwhelming CPU/GPU demand of VR rendering on mobile devices into (1) overwhelming demand on the network bandwidth that is far beyond today's wireless capability, which results in long latency in transferring the rendered frames; and (2) network latency delay from sending rendering request, and receiving the first byte of the rendered frame. Under remote rendering, the time to render a frame is:

$$T_{remote_render} = T_{req} + T_{server_render} + T_{transfer}. \quad (2)$$

To quantify the resource usage and latency achieved of such an approach, we developed and profiled remote rendering versions for the three high-quality, open-source apps in Table 1. Specifically, we used Google Daydream SDK to split each app into a client app and a server counterpart where the client app running on the phone sends new user pose and controller interactions to the server app running on the PC, and the server app performs rendering of all objects and sends the rendered new frames back to the client app over the wireless network.

In our experiment, we ran the server of each app on a strong desktop, which has an Intel 6-core i7-6850K CPU, EVGA GTX1080 graphic card, and 256 GB SSD, and ran the client of each app on PIXEL, and the server sends rendered frames via an 802.11ac WLAN. We used 802.11ac because it provides the highest wireless bandwidth (about 400 Mbps in TCP) on current commodity smartphones.

The results, shown in Table 2, identified two QoE bottlenecks in supporting mobile VR apps using remote rendering over today's wireless networks:

1) Long transfer delay $T_{transfer}$. We measured the QoE when the server streams raw frames in uncompressed format (Bitmap). The three VR apps use up the available network bandwidth (367-384 Mbps) yet still can only achieve on average 5 FPS, and Table 2 shows the latency breakdown in each step. This is because transferring the raw data of one high-resolution frame (i.e., 2560*1440 pixels, about 10 MB) over 802.11ac already takes between 208-216 ms. The per-frame delay significantly elongates the latency and degrades user experience. To support 60 FPS or higher requires streaming each high-quality frame well under 16 ms, and would require at least 4 Gbps or 10X that of today's WiFi bandwidth, or about 200X that of today's LTE bandwidth (which is about 20 Mbps).

We note a recent study [13] also confirmed such high bandwidth requirement of VR apps proposes to use mmWave to accomplish such high bandwidth for VR. However, the solution requires deploying dedicated reflectors and suffers other limitations of highly directional nature of mmWave supporting one user at a time.

2) Network latency delay T_{req}. Even if the wireless network has infinite bandwidth and hence the transfer time is insignificant, sending a request for the new frame and receiving the rendered frame from the server suffers a minimum round-trip delay. To measure the delay, we cannot use the ping utility which would suffer the periodical SDIO (Secure Digital Input Output) bus sleep delay or Power Saving mode delay [22]. Instead, we wrote a simple benchmark that performs back-to-back RTT measurements by sending and receiving 20-byte requests and replies from the server (to avoid the periodical SDIO bus sleep which inflates RTT [22]). Using the benchmark, we measured the RTT and Fig. 3 shows the CDF of the measured RTTs from the PIXEL to the LinkSys EA6350 AP over 802.11 b/g/n/ac, respectively. We see that the latency varies between 2 ms minimum to 4 ms at the 80th percentile, with a long tail reaching beyond 6 ms. These results are consistent with those observed in [22]. The RTT can be explained by about 2 ms base latency of PHY layer transfer (RTS/CTS/DATA/ACK) and the 1 ms (median) delay in traversing the hardware/driver/kernel stack on the phone, which is again limited by the phone hardware capability.

In summary, on today's high-end handset and highest bandwidth wireless network 802.11ac, the time to render a

TABLE 2
Latency Breakdown When We Stream High-Quality VR Content from a Strong Desktop to a Smartphone via 802.11ac WLAN

App	Server Render	Network Delay (req/Xfer)	Frame Size
Viking	11 ms	3/208 ms	10 MB
Corridor	10 ms	3/216 ms	10 MB
Nature	8 ms	3/214 ms	10 MB

The average FPS of each app is about 5. The peak network throughput is about 400 Mbps.

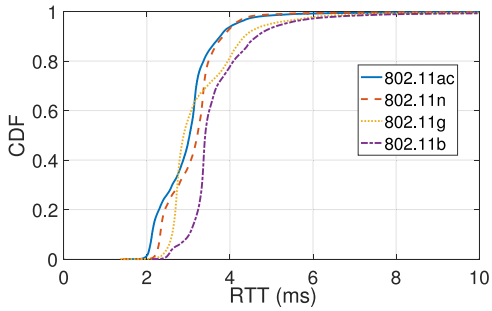


Fig. 3. App-level RTT in different WiFi networks.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

frame under on-demand remote rendering is:

$$\underset{(230\ \text{ms})}{T_{remote_render}} = \underset{(3\ \text{ms})}{T_{req}} + \underset{(11\ \text{ms})}{T_{server_render}} + \underset{(216\ \text{ms})}{T_{transfer}}. \quad (3)$$

Summary. Supporting mobile VR on today's mobile hardware and wireless networking is *fundamentally delay limited* due to the wide gap (at least 10X) between the strict QoE requirements of VR apps dictated by human biology (i.e., visual perception) and the long rendering delay due to limited mobile CPU/GPU hardware and wireless network bandwidth.

2.4 Future Hardware/Networks is Unlikely to Directly Support Untethered High-Quality Mobile VR

Since we have demonstrated that neither local rendering nor remote rendering can support untethered VR on today's mobile hardware and wireless networks, we ask if waiting for tomorrow's mobile hardware and wireless networks can solve the problem. However, as we have shown in [5], we argue that the CPU/GPU of mobile handsets will not grow significantly more powerful because of the power constraint. In addition, the multi-Gbps bandwidth promised by the next-generation wireless networks will not make remote rendering readily feasible because the packet processing will exhaust the mobile CPU.

Given that only waiting for the future mobile hardware or next-generation wireless networks is unlikely to directly help, we argue that the likely path to support untethered high-quality VR apps is via software innovation. In fact, following the same reasoning above, if such a software solution exists, it should work on today's mobile hardware and wireless networks.

3 Exploring Video Compression

We start our quest for a software solution to support untethered VR apps today with the remote rendering approach discussed in Section 2.3.2, by exploring a classic technique for trading off computation time for reduced network transfer time: compression.

It is well known that for transferring consecutive frames that exhibit significant redundancy, as is the case with consecutive frames in a VR app, video compression gives high compression ratio than image compression such as JPEG. Typically, a compressed video is composed of three types of frames: Intra-coded I-frames, Predicted P-frames, and Bi-directional Predicted B-frames. In particular, an I-frame is relatively large and self-contained, and encodes all of the visual information needed to display its content. On the other hand, P-frames are smaller and contain references to prior frames in the stream. To display a P-frame, the client must have the P-frame and any other frames it references. Higher compression ratio of video compression is achieved by encoding the reference frame into an I frame and encoding the subsequent frames as much smaller, dependent P frames.

Recall the original remote rendering approach suffers about 200 ms transfer delay due to the large frame size, about 10 MB. We implemented three state-of-the-art video compression schemes: MJPG (a JPEG-based video encoding scheme), H.264 [23] and VP9 [24], to encode each on-demand requested frame rendered at the server relative to a reference frame previously rendered which is already streamed to the phone. Specifically, in each implementation, we use the mjpeg tool [25], libx264 [23] or libx-vp9 [26] to encode the frame on the server and leverage Android MediaCodec [27] to decode frames on the phone.

TABLE 3
Latency Breakdown When We Stream High-Quality VR Content from a Strong Desktop to a Smartphone via 802.11ac WLAN, with Video Compression

App (Format)	Server (render/ encode)	Phone decode	Network Delay (req/Xfer)	Frame Size
Viking (M)	11 ms/16 ms	134 ms	3/11 ms	407 KB
Corridor (M)	10 ms/17 ms	128 ms	3/11 ms	379 KB
Nature (M)	8 ms/17 ms	137 ms	3/10 ms	369 KB
Viking (H)	11 ms/9 ms	15 ms	3/8 ms	286 KB
Corridor (H)	10 ms/10 ms	16 ms	3/8 ms	277 KB
Nature (H)	8 ms/9 ms	14 ms	3/7 ms	272 KB
Viking (V)	11 ms/193 ms	72 ms	3/7 ms	228 KB
Corridor (V)	10 ms/188 ms	83 ms	3/7 ms	209 KB
Nature (V)	8 ms/202 ms	67 ms	3/6 ms	215 KB

M:MJPG. H:H.264. V:VP9. In each case, the peak network throughput is about 400 Mbps.

We then reran the three high-end VR apps on Pixel and measured the QoE when the server streams compressed frames in MJPEG, H.264 or VP9. Table 3 shows that (1) MJPG compression incurs an encoding time of up to 17 ms and reduces the average frame size to about 400KB for the three VR apps which cuts down the frame transfer time to 14 ms, but it also incurs a high decoding delay of 128-137 ms on the phone, which makes the end-to-end delay in remote rendering a new frame on-demand stay at about 179 ms (3 ms + 11 ms + 17 ms + 11 ms + 137 ms). (2) Although VP9 achieves the highest compression ratio and smallest per-frame size, it also has the highest encoding/decoding time. (3) In contrast, H.264 compression not only cuts down the frame size to be under 286KB and the server encoding time and frame transfer time to be under 10 ms and 8 ms, it also cuts down the phone decoding time to be under 16 ms. Overall, H.264 presents the most effective tradeoff between network bandwidth reduction and phone decoding time.

Under H.264 compression, the end-to-end time to remotely render a frame on demand for the three apps is bounded by:

$$\begin{aligned}\underset{(48\ \text{ms})}{T_{remote_render}} = \underset{(3\ \text{ms})}{T_{req}} + \underset{(11\ \text{ms})}{T_{render}} + \underset{(10\ \text{ms})}{T_{encode}} \\ + \underset{(8\ \text{ms})}{T_{transfer}} + \underset{(16\ \text{ms})}{T_{phone_decode}}.\end{aligned} \quad (4)$$

Examining each of the five components of the end-to-end delay of 48 ms, we see none of the components will easily improve: while higher bandwidth wireless network will reduce $T_{transfer}$, its effect will likely be limited by the CPU capability of the phone as discussed in [5]. Therefore, although the end-to-end delay of 48 ms is only 3X from the per-frame interval of 16 ms, the gap appears very challenging to bridge.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

4 Cooperative Rendering

4.1 Phone/Server Cooperative Rendering

The above insights on the predictability and workload in rendering interactions and environment in VR suggest an effective way to enable high-quality mobile VR is to:

1) divide VR rendering workload into foreground interactions and background environment,
2) leverage the local mobile GPU to render foreground interactions,
3) leverage the remote rendering engine on the server to pre-render and pre-fetch frames of the background environment, and finally,
4) combine them on the phone to generate the final frames.

The key advantages of such a *cooperative rendering approach* between the phone and the server are (1) offloading a majority of the rendering load significantly reduces the computation overhead on the phone; (2) pre-rendering and pre-fetching environments hide the delay from on-demand rendering of environments (first four components in group 1 in Equation (4)); (3) rendering interactions locally avoids the network latency from on-demand rendering of interactions (which cannot be pre-rendered due to unpredictability).

With such cooperative rendering, the time to render a new frame is reduced to:

$$\begin{aligned} T_{co_render} &= max(T_{phone_render_intr}, T_{phone_decode_env}) \\ &\quad + T_{integrate} \\ \text{Constraint:} &\quad \text{environment frames are pre-fetched in time.} \end{aligned} \tag{5}$$

The split architecture design requires an integration step that integrates the locally rendered interactions with remotely rendered environment into the final frame for displaying. Following the typical rendering framework (e.g., OpenGL ES), where the pixel data of each object are rendered and integrated in a frame buffer before being sent to display, in cooperative rendering, the pixel data from the interaction renderer and the prefetched environment frame will be integrated in the frame buffer. Since the integration boils down to memory copying of one frame, it takes less than 1 ms on the phone.

Thus, barring the following two remaining challenges, the above cooperative renderer design holds the promise to render a frame within the per-frame rendering interval of 16 ms:

Q1: How to pre-render and pre-fetch environment frames just in time to be used for rendering the new frame?

Q2: How to speed up phone decoding time to be within the 16 ms interval?

4.2 Prefetching Panoramic Frames

In the basic cooperative renderer design, we assume the next frame to be rendered can be pre-rendered and pre-fetched. However, straightforwardly applying prefetching in VR would face two challenges: (1) There are almost infinite possibilities for the next pose since a user can freely change position and orientation; (2) Delivering a large number of high-quality frames pre-rendered by the server to the client will incur significant bandwidth overhead and accordingly high CPU utilization on the client (from packet processing).

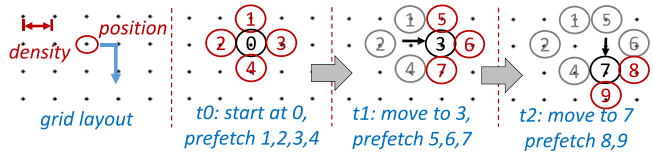


Fig. 4. The prefetching process.

We tackle these challenges with a set of VR-specific optimizations as follows.

Leveraging Movement Delay to Prefetch Frames. In a VR system, the virtual world is discretized into grid points or grid locations, as shown in Fig. 4, and rendering the faraway environment at any location in between grid points is approximated with rendering from the nearest grid point. In particular, the entire grid size is VR app-specific, and scales to the size of VR map (e.g., a larger VR map may have larger grid size). We further define the distance between two adjacent positions as *grid density (d)*. Effectively the grid density determines the smoothness of rendering during user movement in the virtual world, and should be carefully designed. A too coarse-grained grid may lead to some jumpy changes in the background when the user moves quickly, while a too fine-grained grid can incur significant additional rendering overhead. Previous studies (e.g., [11]) have shown that for high-end VR apps, setting the density to 0.02 Unity units (which corresponds to about 2 cm) is sufficient to sustain smooth rendering during user movement. Hence we set grid density $d = 0.02$ in our design.

The discretization of the virtual world and delay in user movement to the next grid location provides us the opportunity to prefetch the frame for the next location. In particular, the time it takes the user to move from one grid point to an adjacent point is determined by the movement speed v. We make an observation that for realism, the typical user movement speed (e.g., walking) in the virtual world is similar to in the real world, at about 1m/s. Dividing the grid density by this speed gives 20 ms as the time interval for a user to move between adjacent grid points. This is the time window for prefetching the frame for the next grid location.

Using a Panoramic Frame to Encapsulate all Possible Orientations. In a VR system, a user pose is composed of a position and an orientation. While there are only a few adjacent grid locations the user can move to next, at the next grid location, the user can freely change to any orientation (by turning the headset) abruptly which is difficult to predict, and hence pre-rendering and prefetching the frame with the right orientation is hard.

To overcome this challenge, we pre-render a *panoramic frame* for each grid location which conceptually aggregates or encodes all possible orientations at that fixed position, i.e., a frame with any orientation from that location can be easily cropped from the panoramic frame. Whenever the user moves to one location, we can start prefetching the panoramic frame for the next grid location.

Prefetching Frames for all Adjacent Positions. Since there are four grid locations surrounding each grid location, prefetching one panoramic location for only one adjacent grid location requires predicting which grid location the user

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Cutting Down Pre-Rendered and Pre-Fetched Frames

Method	Data to fetch (In a time window (d/v))
Naive prefetching	5*N frames (5*10*N MB)
Panorama	$\leq$ 3 panoramas (3*31 MB)
Panorama+ +Video compression	1 video (300-400 KB)

will next move to, which is not always easy. To avoid complicated prediction and miss-predictions, at each grid location, we prefetch panoramic frames for all adjacent grid locations.

Since prefetching is done incrementally as the user moves, the number of frames to prefetch at each grid point is up to three, as shown by the example in Fig. 4. Assume initially the user stands at position 0, and panoramic frames 1-4 are already prefetched. Then at t1 the user starts to move to position 3, and at the same time we prefetch panoramic frames for positions 5, 6, and 7. Similarly at t2, the users starts moving to position 7 and only frames in 8 and 9 are prefetched. Note in moving between positions 3 and 7, since the phone already prefetched panoramic views for both positions, it can decode a frame with any orientation locally.

Encoding Multiple Panoramic Frames. By default, assume that there are N possible orientations in a certain position, we have to prefecth 5*N possible frames. To cut down the frame amount, we use a 360 degree panoramic frame that covers all possible orientations to batch all frames in the same position. Therefore, when the user quickly rotates the headset but the position does not change, we can quickly generate a final frame from the panorama, without fetching a new frame from the server. Fetching a single panoramic frame for each grid location instead of many possible frames (for N possible orientations) already significantly cuts down the total frame size, from 5*N*10 MB down to 3*31 MB when without video compression, as shown in Table 4.

Further, We explore video encoding to compress multiple panoramic frames. Encoding the three panoramic frames incrementally (i.e., into three dependent P-frames) potentially gives higher compression ratio, but this can significantly increase the decoding time, to about 110 ms, as decoding the last of the three panoramic frame in the video requires decoding the first two. To remove the above dependency of any P-frame on the preceding I frame and P-frames, one approach is to encode each frame as an I-frame which can be decoded independently [11]. However, an I-frame has a larger size than a P-frame and thus reduces the compression ratio.

For this reason, we propose a new non-successive variation of video encoding denoted as *direct encoding* to reduce the latency of decoding P frames. Since in VR systems the virtual world is discretized into grid positions, we divide the entire map into multiple *"sub-areas"*, and each sub-areas is a square with length of L, covering a number of gird positions as shown in Fig. 5. We define the central position in a sub-areas as the *"anchor position"* of the sub-area. In every sub-area, we encode the panoramic frame in the anchor position as an I frame (i.e., the anchor frame) which can be decoded independently, and encode all other panoramic

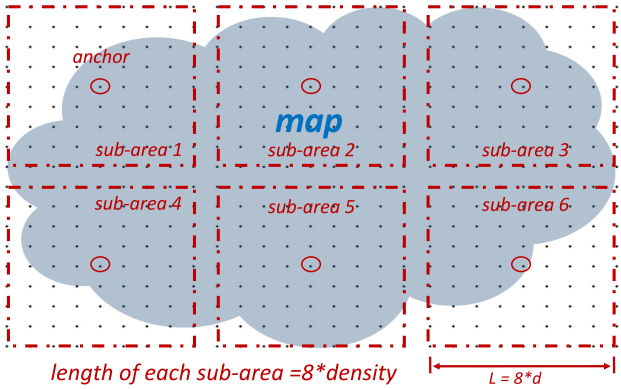


Fig. 5. An example of the sub-area and anchor position in the grid layout. The VR map is divided into six sub-areas and each sub-area is a square of $8 * d$ length.

TABLE 5
Latency of Each Step When We Prefetch the Background Environment for Viking Village

On-demand	$T_{phone_render_intr}$ 13 ms	T_{phone_decode} 45 ms	
Prefetching	T_{req} 3 ms	$T_{transfer}$ 10 ms	T_{walk} 20 ms

Latencies of other two VR apps are similar.

frames in the same sub-area as an independent P frame relative to the I frame in the anchor position. To successfully decode every P frame, the client pre-fetches the anchor frame when moves to a new sub-area. Making all P-frames not dependent this way reduces the decoding time of any P-frame (panoramic frame) to 45 ms on average, at the cost of a slightly decreased compression ratio (about 9 percent reduction). With video compression, one panoramic frame can be compressed down to a P-frame of under 133KB on average, as shown in Table 4.

Delay Reduction. We measured the delay reduction on the three reference VR apps from prefetching panoramic frames and video compression discussed above on the PIXEL phone, over 802.11ac WiFi. Table 5 shows five components of the process. On one hand, sending the request for and transferring of three panoramic frames can be finished in 13 ms, which is less than the 20 ms delay for the user to walk from one grid location to the next, and hence satisfying the prefetching constraint in Equation (5):

$$Constraint: \underset{(3\text{ ms})}{T_{req}} + \underset{(10\text{ ms})}{T_{transfer}} \leq \underset{(20\text{ ms})}{T_{walk_to_next_grid_point}}. \tag{6}$$

On the other hand, decoding any one of the three panoramic frames in the compressed video takes about 45 ms, much higher than the 16 ms for decoding a compressed regular frame (Table 3).

4.3 Parallel Decoding on the Phone

We observe that decoding a panoramic frame is an inherently sequential process, and hence does not employ the

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

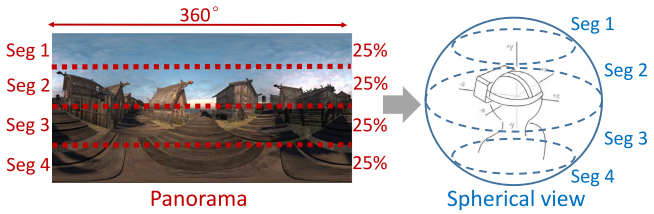


Fig. 6. Parallel decoding: Leveraging multi-core to decode frame segments in parallel.

parallelism offered by the multiple cores on modern handsets. To exploit the hardware parallelism, we cut each panoramic frame into N segments, each at $1/N$ of the original size, encode the corresponding segments of the three panoramic frames into N videos. Since the redundancy remains between the corresponding segments and that of the reference panoramic frame (on the phone), the video compression ratio should remain high. More importantly, decoding on the phone of the N segments of each panoramic frame can now be done on N cores in parallel. Since PIXEL comes with four cores, we experimented with different N (between 2 and 8) and found $N = 4$ gives the best decoding time. Fig. 6 shows how a panoramic frame is segmented into 4 segments.

During the pre-fetching process, the client maintains a LRU-based frame buffer to cache the already decoded frames. If a user moves back to an old grid position where the corresponding panoramic frame has been pre-fetched and decoded in the buffer, the client just read the frame from the buffer directly. Thus, the frame buffer avoids decoding a lot of repetitive frames if the user walks around in the same area.

Table 6 shows parallel decoding of four segments reduces the phone decoding time from 45 ms to 12 ms, and the end-to-end time to render a new frame to $max(13\text{ ms}, 12\text{ ms}) + 1\text{ ms} = 14\text{ ms}$, which allows the VR apps to achieve 60 FPS.

5 Flexible Background Adaptation

So far we have described a series of techniques to enable high-quality immersive virtual reality on today's mobile devices and wireless networks. These techniques require very good network condition. For example the rendering server has to be deployed on a machine placed in the WLAN to provide high available bandwidth and low latency. To further extend the flexibility and applicability of our system, we propose a *flexible background adaptation* scheme to enable high-quality and low-latency VR under various network conditions (e.g., the server is deployed in the cloud) by dynamically adjusting visual quality according to the field of view and prefetching area. Intuitively, the key idea of our scheme can be summarized as: (1) prefetching more nearby reachable frames during the movement to tolerate higher network latency; and (2) properly reducing the resolution of a frame according to the field of view to match the available bandwidth in real time.

The Prefetching Circle. As we have shown in Fig. 4, the virtual world is discretized into grid points. Let the user move in speed v (i.e., the user walks through v grid points per second). Based on the user's current position, we create the *prefetching circle* which represents the area where the user may

TABLE 6
Latency of Each Step with Parallel Decoding for Viking

On-demand	$T_{phone_render_intr}$	T_{phone_decode}
	13 ms	12 ms

reach in next X seconds, and thus the radius of the prefetching circle is $R = v * X$. Once the user moves to a new position, FURION prefetches all corresponding panoramic frames covered by the prefetching circle. Therefore, to sustain good QoE under various RTT and available bandwidth, next we explore two questions: under a certain network condition, (1) how to properly set the radius of the prefetching circle so that future frames can be prefetched and used on time? (2) given a certain bandwidth, how to dynamically adjust the bitrate of panoramic frames inside the prefetching circle with minimum visual quality degradation?

Aggressive Prefetching Scheme. Fundamentally, the goal of prefetching is to download a new frame timely before it is used so that the network latency can be hidden. However, under a higher RTT network condition where the request delay (T_{req}) is longer than the movement delay between two adjacent positions (e.g., $RTT >= 20\text{ ms}$), it is too late if the client only prefetches adjacent frames in the next one step. In this case, the client has to prefetch more frames (i.e., larger prefetching circle) to guarantee frames be used on time. Note that FURION maintains a LRU-based local frame buffer to cache all prefetched frames. Each time the user moves to a new position, FURION sends a request to the server to prefetch all panoramic frames covered by the prefetching circle. The local buffer is updated in every RTT, therefore to guarantee that every time in a new position the corresponding frame is prefetched. FURION dynamically sets the radius of the prefetching circle $R = v * X, (where X = 2 * RTT)$, indicating that the client always prefetches reachable frames in the next two RTTs. For example, in Fig. 4 the movement delay between two grid positions is 20 ms. In t0, if the RTT is 20 ms, the client prefetches all frames reachable in next 40 ms such as frame 6 (i.e., all frames in next two steps).

Panorama Bitrate Adaptation. The aggressive prefetching scheme involves additional bandwidth overhead, and may lead to high interactive latency if the available bandwidth is insufficient. Therefore we propose the bitrate adaptation scheme to adjust the bitrate of panoramic frames streamed to the client to satisfy the real-time network bandwidth, and guarantee that frames can be delivered on time. Intuitively, a naive bitrate adaptation scheme reduces the bitrate of the stream if the bandwidth is insufficient, but it can significantly harm the visual quality and thus lead to poor QoE. Instead, we propose a new adaptation scheme with the goal of reducing bitrate to satisfy the bandwidth while causing minimum visual quality degradation. Our scheme is based on a key fact that given a panoramic frame, the user can only see a portion of the entire panoramic frame. When the bandwidth is insufficient, we first reduce the visual quality outside of the field of view to minimize the QoE degradation.

Typically the FoV of one panoramic frame is $360 * 180$ degrees, and for most existing VR headset the FoV is about $100 * 100$ degrees, (e.g., $100 * 90$ degrees for Daydream [7],

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

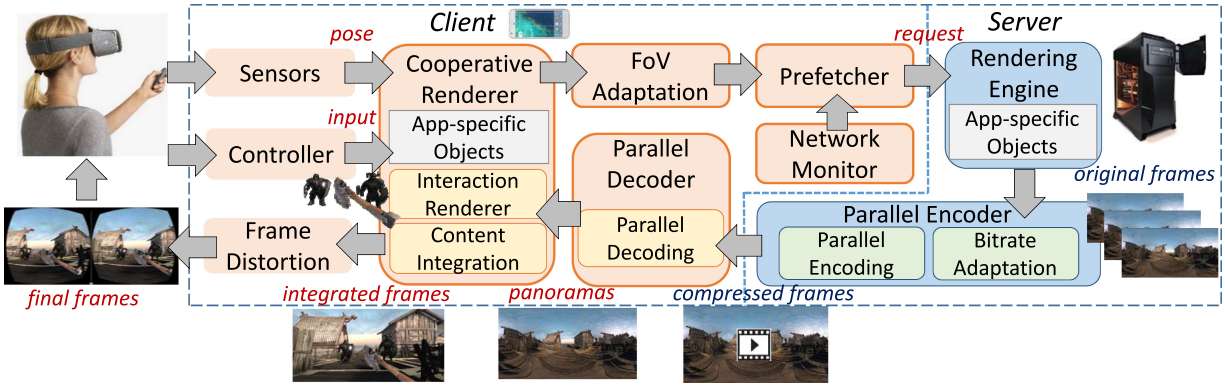


Fig. 7. FURION system architecture.

$96 * 90$ degrees for GearVR [15])indicating that the current FoV viewed by a user can at most cover 2 of the 4 segments as shown in Fig. 6.

Accordingly, FURION pre-renders and pre-encodes each panoramic frame in different bitrates, and under our parallel decoding scheme each panoramic frame is then split into four segments. At runtime if the bandwidth is getting insufficient, the FURION client preferably requests the low visual quality version of the 2 segments outside of current FoV to match the available bandwidth. In addition, we design a network monitor module to monitor the performances of underlying networks.

Algorithm 1. FoV-Based Bitrate Adaptation

```
while true do
    In every RTT
    Estimate current RTT
    Estimate current bandwidth (Segment_size*cwnd)/RTT
    Select bitrate based on FoV and bandwidth
    Prefetch frames in 2 * RTT area
end while
```

Algorithm 1 shows the details of the background bitrate adaptation process. Assume the virtual world is discretized and the user moves from an old position to a new position. At the beginning of each iteration, the client estimates the current RTT by calculating the average value of the past five RTTs. Assume $cwnd$ is the size of current congestion window, and the current bandwidth is estimated as $segment_size * cwnd$. Then the client selects a proper bitrate for each segment based on current FoV and bandwidth. The bitrate of segments out of current FoV is preferably reduced to minimize the degradation of visual quality when the bandwidth is insufficient.

6 PUTTING IT TOGETHER: FURION ARCHITECTURE

Incorporating the above ideas, we have developed FURION, a VR framework that enables high-quality, immersive mobile virtual reality on today's mobile device and wireless networks, by providing high visual quality, high frame rate, and low latency. Next we introduce the system architecture and the support of customized app features.

Architecture and Workflow. Fig. 7 shows the system architecture and the workflow of FURION. At a high level, FURION includes the client part and the server part which cooperatively render VR contents to the user. The sensor module on the client tracks pose changes, including both position and orientation changes of the user headset, together with additional user inputs from the controller. The key component on the client is the *cooperative renderer*, which splits the VR workload into a local part and a remote part, and offloads the remote part to the server. Specifically, for each user pose, it renders locally the interactions and requests the parallel decoder to decode the background environment from a pre-fetched panoramic frame (4 subframes), combines the rendered interactions and decoded environment and sends it to the frame distortion module, which generates the final frame for each eye and displays to the user.

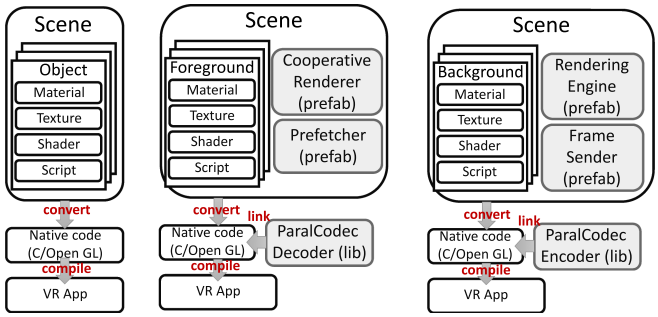
Whenever the user moves to a new grid location, the prefetcher sends a request to the server to request panoramic frames according to current user FoV and network condition measured by the network monitor. The server part is a generic environment rendering engine that is initialized with app-specific objects, and pre-renders panoramic frames for each grid location.

At runtime, the parallel encoder engine loads panoramic frames specified in the client request, cuts each panoramic frame into 4 segments, and encodes them using H.264 codec. The bitrate adaptation module then selects the proper frame bitrate according to real-time network condition, and transfers the encoded video to the phone over the wireless network.

Rendering Dynamic Objects in Background Environment. To build an immersive user experience, in some VR apps the background environment may contain several dynamic objects, e.g., floating clouds in the sky, or flowing creeks in the mountain. These dynamic objects such as the flame in Fig. 2b can be easily extracted in Unity development environment, and they are typically small and have similar workloads as foreground interactions. FURION can support dynamic objects in the environment by rendering them on the local GPU and superimposing them on the rest background environment. We demonstrate FURION scales well to the number of dynamic objects in the background in Section 9.

Synchronizing Interactions and Environment. Typically, VR developers have to define the app logic at the development time. For example, a developer may define the position of a passable road and an obstacle where the player cannot go through. Under FURION's split architecture, such logic requires to synchronize context information between foreground interactions and background environment. FURION

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.



(a) Unity (b) Building the client. (c) Building the server.

Fig. 8. Developing VR apps with FURION.

achieves the synchronization by maintaining a *property map* in both the client and server. The property map records the $< position, property >$ pair of each grid in the virtual world, where *property* is a developer-defined attribute of the grid point. At the development time, the developer can configure the property map to set the reachable area or any app-specific event trigger in certain positions.

7 DEVELOPING VR APPS WITH FURION

We have implemented FURION on top of Unity and Google Daydream SDK. Before discussing the implementation details, we first describe the user interface of FURION, i.e., how a developer uses FURION to develop VR apps.

Quick Primer of Unity. Typically a VR app is developed by some virtual scene creation tools or game creation IDE, such as Unity and Unreal. We build our FURION system on Unity. Fig. 8a shows the basic workflow of building apps with Unity. Specifically, Unity provides a WYSIWYG editor that allows a developer to work on a *scene* that describes the interactions and virtual environment in the app. A scene contains a number of objects that represent the basic unit at the development time, and the developer can change the property of an object by adding new material and texture to it. In addition, the developer can also write a script for each object to add animation and game logic. Unity then converts the visible scene to native codes and finally compiles the code to generate the executable app.

Developing VR with FURION. FURION provides an easy-to-use way for developers to augment VR apps developed using Unity. Figs. 8b and 8c show the process to build the client and server part of a VR app with FURION. Specifically, we implement and export the cooperative renderer, the prefetcher and the rendering engine as specific prefabs in Unity, which can be imported and used in the app at the development time. In addition, we implement the parallel codec module as a library which is linked at the compiling time. For developers, augmenting a VR app with FURION only requires adding FURION prefabs in the scene and configuring app-specific parameters (e.g., assigning foreground and background objects, or setting the reachable area in the scene) in the prefab.

8 IMPLEMENTATION

We describe a number of practical issues in implementing FURION.

Cooperative Renderer. In FURION, the cooperative renderer renders interactions using the mobile GPU and fills the surrounding environments with the panoramic frame rendered by the server. We leverage the native Unity API to write a Unity prefab called *EnvTexture* that can load frames generated from an external player and draw surrounding environments. With EnvTexture, FURION composes interactions and environments rendered on the client and server respectively. The cooperative renderer is implemented in around 1900 lines of C# code.

Prefetcher. We implement the prefetcher module in FURION as a prefab in Unity to calculate the user position and determine the time and data of a prefetching operation, according to the current user position. We build a series of rendering cameras in the rendering engine on the server to pre-render panoramic frames, each containing 3840*2160 pixels. The prefetcher together with the rendering engine on the server are implemented in around 2100 lines of C# code.

Parallel Codec. The basic requirement of our Parallel Codec is to encode the 4 segments per panoramic frame on the server and to quickly decode them to display a specific frame on the mobile client. We implement the parallel codec module by extending existing ffmpeg [28] and x264 [23] libraries. We extend the encoding/decoding module in the x264 library to support direct encoding and modify the sequential playback logic in ffmpeg to support parallel decoding. To synchronize all segments during the playback in ffmpeg, we build one decoding thread for each segment and a master thread to send the entire frame to the display hardware once all segments are decoded. In addition, we build a separate network process to deliver the compressed video to the client over TCP. Collectively, we modified about 1100 lines of C code in the ffmpeg and x264 libraries.

Bitrate Adaptation. We implemented the server part of the bitrate adaptation module on a Nginx server. More specifically, we set four bitrate options (i.e., 40 Mbps, 20 Mbps, 10 Mbps, 5 Mbps) for each segment of a panoramic frame. At runtime, the rate adaptation module monitors network condition and selects the best bitrate based on both network condition and user's FoV to sustain acceptable QoE.

Sensing and Distortion. We leverage the Google Daydream SDK to (1) build the sensor module to read pose and additional user inputs from the controller and (2) perform distortion on the frame generated by the cooperative renderer.

9 PERFORMANCE EVALUATION

In this section, we evaluate the end-to-end performance and system overhead of VR apps built upon FURION.

9.1 Experiment Setup

We run the FURION server on a strong desktop as we described in Section 2. We run the client app on a PIXEL smartphone running Android 7.1 with Google Daydream support. The client communicates with the server via an 802.11ac AP which supports up to 400 Mbps TCP throughput for a single device. We augment the three high-quality apps as we described previously (Viking [8], Corridor [9], and Nature [10]) with FURION to support high-quality virtual reality.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 7
Visual Quality and FPS of Each VR App under Different Implementations

App (Implementation)	Visual Quality (Average SSIM)	Average FPS
Viking (Mobile)	0.812	11
Corridor (Mobile)	0.834	9
Nature (Mobile)	0.833	15
Viking (Thin-client)	0.927	36
Corridor (Thin-client)	0.933	34
Nature (Thin-client)	0.944	37
Viking (FURION)	0.927	60
Corridor (FURION)	0.933	60
Nature (FURION)	0.944	60

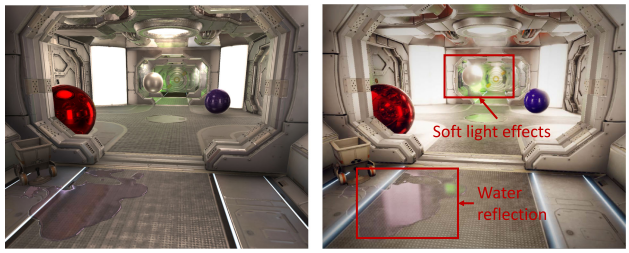
We build three implementations of each app in our experiment: (1) *Mobile*: a local rendering version that renders all VR content locally; (2) *Thin-client:* a remote rendering version that offloads all rendering workloads to the server and encodes frames with H.264; (3) FURION: built on FURION which leverages both client and server to cooperatively render VR content. We do not compare our solution with MoVR [13], [14] because there is no 60 GHz WiFi support in current commodity smartphones, or with FlashBack [11] which does not support user interactions.

In the FURION versions of the three VR apps, the total size of the panoramic frames pre-rendered by the server are 74 GB, 71 GB, and 53 GB, respectively. Note that because FURION sets a static gird density (0.02 Unity units) and pre-renders a panoramic frame in every grid position, the size of the panoramic frames pre-rendered by the server scales linearly to the original map size in the three VR apps. These pre-generated panoramic frames are stored on the server, and do not increase the storage overhead on the mobile VR headset.

9.2 Visual Quality, Frame Rate, Responsiveness

Image Quality. We use Structural Similarity (SSIM) [29] to quantify the image quality of different schemes. SSIM is a standard metric from the video processing community used to assess the user-perceived loss in frame quality between the original version A and a distorted version B. SSIM has a value between 0 and 1; a larger SSIM value indicates higher fidelity to the original frame. In our experiment, we use SSIM to compare each final frame rendered by each scheme with the corresponding original frame rendered by the strong desktop (before encoding). All final frames are rendered with 2560*1440 pixels.

Table 7 shows the visual quality results. The SSIM value of the mobile rendering system is about 0.812, which falls short of high graphics quality. The visual quality decreases because the quality manager inside the Android rendering system automatically disables some functions that provide immersive visual effects (e.g., anti aliasing and color grading) but are too computation intensive on mobile devices. The SSIM values under FURION and Thin-client are identical as both go through H.264 encoding and decoding, and integration of rendered interactions and decoded environment under FURION does not affect image quality. Further, the SSIM values are above 0.927. A SSIM value higher than 0.9 indicates that the distorted frame well approximates the original high-quality frame and provides "good" visual quality [30].



(a) Mobile rendering. (b) FURION rendering.

Fig. 9. The frame for the same pose rendered by the mobile system (left) and FURION (right). FURION achieves higher SSIM and provides more immersive experience (e.g., the immersive soft light effects and water reflection rendered in the right figure).

More intuitively, Fig. 9 shows an example of the same frame of Corridor rendered by the mobile version (the left one) and rendered under FURION (the right one). The visual quality of mobile version is inferior that FURION. This is because rendering high quality VR contents on the phone involves significant computation overhead, functions such as anti aliasing and color grading are disabled.

Frame Rate. To enable a smooth immersive experience, it is necessary for the VR system to deliver at least 60 FPS, a frame rate as high as the headset supports. Table 7 plots the frame rate of three different VR implementations running the three VR apps. On average, the mobile system delivers the lowest FPS, between 9-15, indicating that the constrained mobile GPU/CPU is insufficient to deliver a satisfactory frame rate. The thin-client version delivers which offloads all rendering workload to the server gains between 34-37 FPS, which is mainly limited by the RTT and the decoding capability on the client. FURION comfortably delivers 60 FPS, because we properly place the rendering workload on the client and server to enhance the rendering capability for the mobile device. We note that the 60 FPS frame rate of FURION is not limited by our implementation, but by the 60 Hz refresh rate of the display hardware on PIXEL.

Responsiveness. Following [11], we define app responsiveness as the latency or elapsed time from when the last pose or user action is received from the device until when the corresponding frame is sent to the display. We note the latency thus defined does not include the latency contributed by the input subsystem and the display hardware, which are difficult to measure and equally present when VR apps execute under any of the three schemes. We evaluate the app responsiveness under three user interaction scenarios: (1) *controller latency* for clicking the controller; (2) *rotation latency* for rotating the headset (orientation); and (3) *motion latency* for changing the position.

Fig. 10 shows FURION achieves far better responsiveness than the other two systems. In particular, it achieves under 14 ms controller latency, under 1 ms rotation latency, and under 12 ms motion latency for the three apps, respectively. In FURION, the rotation latency is much lower than the controller and motion latency because FURION can quickly generate a final frame for a given orientation by cropping the in-memory, already decoded panoramic frame.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

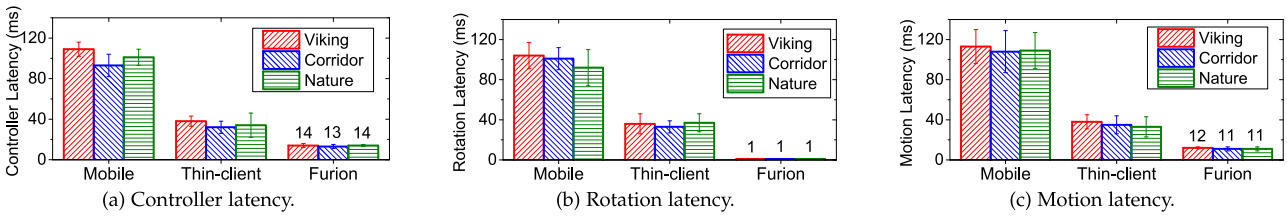


Fig. 10. Responsiveness of different VR implementations.

9.3 Scalability with App Features

Next, we evaluate the scalability of different schemes in terms of different app features. Typically, both foreground interactions and background environment in VR apps contain dynamic objects to create an immersive experience. Different apps may have different the number of dynamic objects, in the foreground interactions and background environment, which enhances the immersive experience of the VR apps.

We modify the three VR apps with an increasing number of dynamic objects in foreground interactions and measure the frame rate of each implementation. Figs. 11 and 12 shows the result for Viking Village in terms of dynamic objects in foreground interactions or background environment respectively. Results of the other two apps are similar and omitted due to the page limit. We see as the number of dynamic objects increases, the local rendering workload increases and as a result the FPS of the mobile version decreases. The FPS of the thin-client version does not change because adding more dynamic objects only increases the workload on the server. Finally, FURION can maintain high FPS when supporting up to 10 foreground interactive dynamic objects or up to 30 background dynamic objects. FURION can support more background dynamic objects because background objects are usually remote and incur relatively lighter rendering overhead.

9.4 Resource Usage

We next evaluate the resource usage of FURION in supporting the three VR apps.

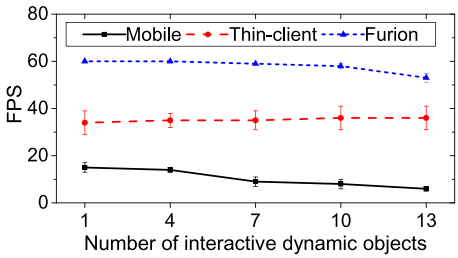


Fig. 11. FPS as the number of interactive dynamic objects varies.

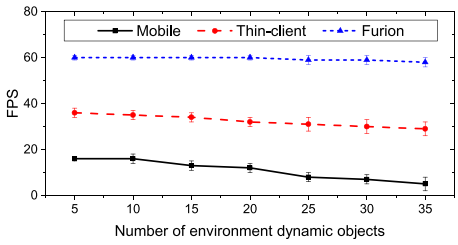


Fig. 12. FPS as the number of background dynamic objects varies.

CPU/GPU Usage. We first measure the CPU/GPU utilization together with the network usage in Table 8. We see FURION incurs up to 65 percent CPU utilization and 37 percent GPU utilization on the PIXEL phone in supporting the three high-quality VR apps with target low latency and 60 FPS.

Since Google labeled the PIXEL phone as VR-ready with its introduction [4], we were curious what new capability made the most difference, compared to a previous generation phone, Nexus 6P. We evaluated FURION on Nexus 6P and compared its resource usage with that on PIXEL. Table 8 shows that running the three apps on Nexus 6P incurs much higher CPU and GPU utilization, up to 76 percent for CPU and 64 percent for GPU, suggesting the VR-readiness of PIXEL mainly comes from the upgraded GPU.

CPU Usage and Temperature over Time. Today's high-end phones such as PIXEL have built-in temperature control to avoid phone overheating. For example, by default the thermal limit of PIXEL is 58 Celsius (from reading /system/etc/thermal-engine.conf on the phone). When the thermal limit is reached, e.g., from long running, power-intensive apps, the power control manager will cap or reduce the CPU/GPU frequency to avoid overheating the phone.

Since VR apps present a new class of power-intensive apps, and in fact Google proposed explicit thermal requirement [31] that "your Daydream app must not hit the device's thermal limit during 30 minutes of usage", we measured the phone SoC temperature when running the three VR apps under FURION for 30 minutes. Fig. 15a 15b shows during the 30-minute runs, the CPU frequency and utilization remain steady, and the SoC temperature increases gradually but stays under the thermal limit.

Power Drain over Time. We leverage the Google Battery Historian tool [32] to inspect the power drain of the Android device over time. Fig. 15c shows that the power draw stays steady at 410 mA on average in running the three VR apps over the extended half-hour period. Given PIXEL's battery capacity of 2770 mAh, at such a power level, the three VR apps can last for about 6 hours.

Memory Overhead. Since FURION prefetches panoramic frames and caches them in a local buffer, the client of FURION

TABLE 8
System Overhead of VR Apps upon FURION

App	CPU Utilization (Pixel/6P)	GPU Utilization (Pixel/6P)	Average Bandwidth (Pixel/6P)
Viking	64% / 76%	37% / 63%	127 Mbps / 131 Mbps
Corridor	62% / 71%	34% / 61%	132 Mbps / 122 Mbps
Nature	65% / 73%	33% / 64%	130 Mbps / 128 Mbps

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

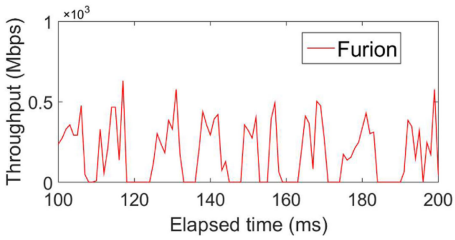


Fig. 13. Bandwidth usage in each 20 ms window.

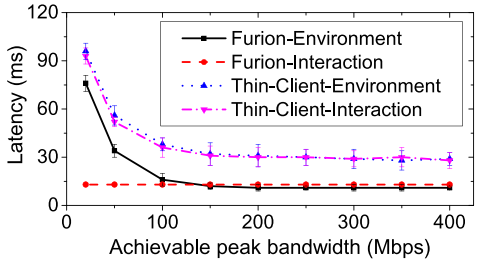


Fig. 14. Rendering latency under bandwidth capping.

occupies addition memory overhead. In our current implementation, FURION takes about 30 MB more memory than the thin-client implementation, which is a doable overhead in today's commodity mobile devices.

9.5 Robustness to Network Instability

Table 8 further shows that supporting the three VR apps in FURION requires on average 132 Mbps network throughput, well below the 400 Mbps peak throughput supported by 802.11ac. Fig. 13 shows this is because prefetching panoramic frames of adjacent positions in the virtual world can quickly complete in a short burst (about 8 ms per Equation (4)) within each 20 ms interval of moving between two adjacent positions. The throughput value in Fig. 13 starts from 100 ms to exclude the slow start stage.

In fact, the cooperative renderer design of FURION which turns on-demand fetching of frames from the server into prefetching pre-rendered frames, effectively moves fetching frames from the server outside the end-to-end delay path of rendering a new frame. As a result, prefetching pre-rendered panoramic frames just needs to finish within the 20 ms window. This property suggests FURION can tolerate transient bandwidth fluctuations as long as the network provides 130 Mbps bandwidth on average (in each window). To confirm this property, we used the `tc` tool to control the maximal available bandwidth on the AP and ran the three VR apps. Fig. 14 shows under FURION the delay in rendering of interactions and in decoding environment stays constant when the average peak bandwidth is reduced from 400 Mbps to 150 Mbps.

9.6 Performance of Background Adaptation

So far we have demonstrated that FURION is able to enable high-quality immersive VR on commodity mobile devices and WiFi network. Since we have designed the flexible background adapatation scheme in FURION to wisely adjust the bitrate of prefetched panoramic frame, FURION can also provide acceptable QoE under different network conditions (e.g., if the server is deployed on a remote server instead of a nearby machine). Next, we evaluate the achievable QoE of FURION under various network conditions. We use `tc` to adjust the network RTT and maximum bandwidth to simulate various network conditions. To evaluate the capability of adapting to various network conditions, we measure the average latency of three background transmission schemes: (1) *Thin-client:* fetching required frames on-demand, without prefetching and bitrate adaptation; (2) FURION without bitrate adaptation; and (3) FURION implemented with bit adaptation described in Section 5. The latency is measured as the elapsed time from when the last user pose is received from the device until the time when corresponding frame is sent to the display.

Fig. 16 plots the latency of typical VR operations (e.g., motion or playing with controller) and visual quality of each implementations under various network conditions. We only show the result of Viking Village app and the other two apps are similar and omitted due to the page limit. For VR implementations without background bitrate adaptation, the SSIM value keeps high but the frame latency significantly increases as the network RTT increases or bandwidth decreases, which leads to unacceptable user experience. FURION implemented with FoV-based bitrate adaptation achieves the lowest frame latency while keeping high SSIM (e.g., > 0.91) under different network conditions.

10 Limitation and Discussions

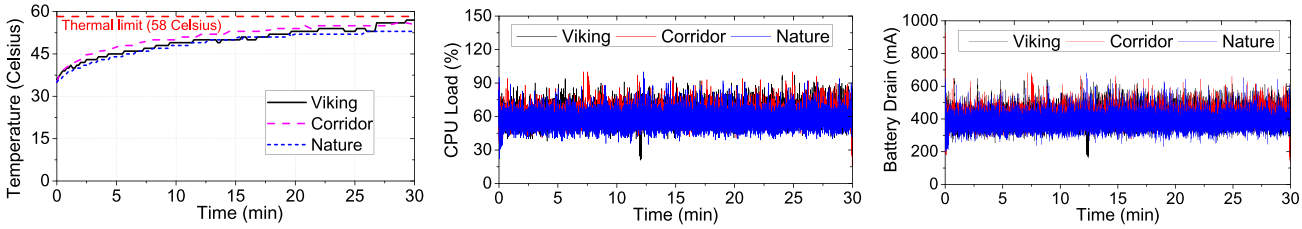
We now discuss the limited usage scenario of FURION and how FURION design can be extended to handle more complicated VR app features.

Limitation on Dynamic Lighting Models. Since FURION separately renders foreground interactions and background environment, it is more limited on supporting dynamic lighting model in which the light source dynamically changes based on user interactions (e.g., a user illuminates the room with a flashlight). This is because the properties of light source (e.g., color or direction of the light) affects the appearance of shadow and reflection. If the light source is interactive, then shadows and reflections are interactive and rendered as foreground interactions which are limited by the capability of mobile GPU, as shown in Fig. 12.

Limitation on Rendering Complex Dynamic Objects. FURION is limited to support VR app with very complex dynamic objects that are closely related to the background environment. This kind of dynamic objects may be blended into the background based on real-time user's position (e.g., a flashing film screen), or interact with the background (e.g., an explosion on the top of the background).

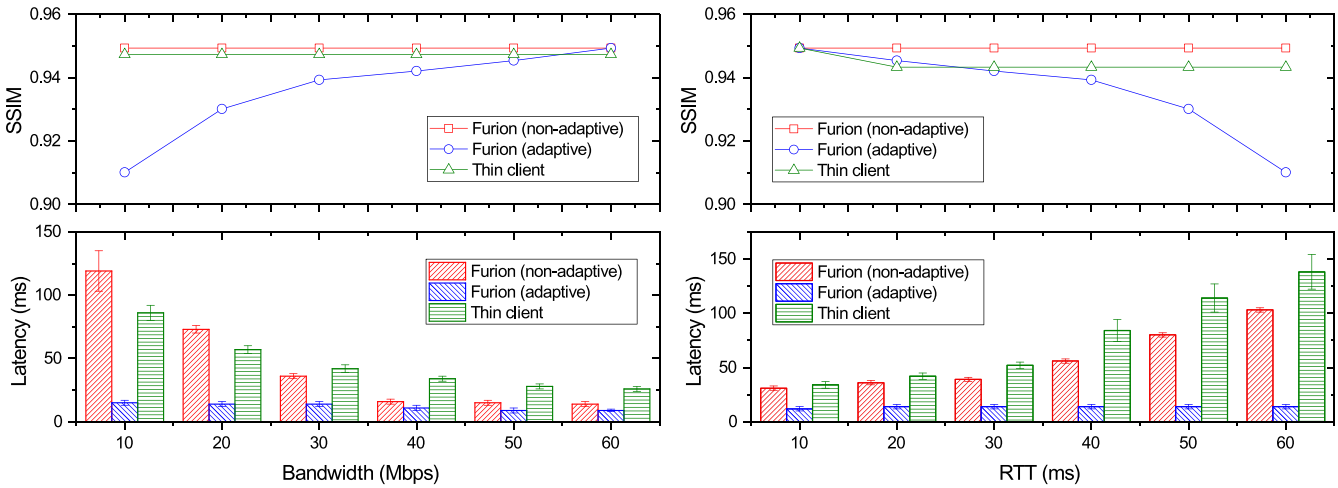
Handling Drastic Movement. In a VR system, the virtual world is discretized into grid points and FURION leverages the movement delay between adjacent points to prefetch future frames in time. As discussed in Section 4.2, the movement delay is about 20 ms under a normal user movement. However, there can be cases where the user moves more quickly and the movement delay is too low to prefetch the adjacent panoramic frame in time. FURION does not change the gird size or density under various movement speed. In practice, such drastic user movement implies there is no need to render frames at consecutive grid positions, and it can be handled by selective-prefetching, i.e., prefetching frames from further grid

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.



(a) Temperature of three VR apps. (b) CPU utilization over time. (c) Battery power.

Fig. 15. Resource usage of FURION over time. During the experiment, the CPU frequency of all four cores is capped by the default CPU governor of PIXEL at 1.44GHz. The thermal limit on Pixel is 58 Celsius.



(a) Frame latency under varying available bandwidth. (b) Frame latency under varying network latency.

Fig. 16. Frame latency and SSIM of each VR system under various RTTs and maximum bandwidths.

points. For example, at moment t1 in Fig. 4, if the user moves at twice the normal speed, we just need to prefetch the frames from 2 grid positions away.

Multi-Player Support. Some latest VR apps such as vTime [19] and Altspace VR [33] contain social elements and enable multiple users playing in the same virtual environment, e.g., chatting in the same office. A key feature of multi-player VR apps is that some foreground interactions are updated according to inputs (e.g., pose or controller changes) from other players. In our future work, we plan to extend the split renderer of FURION to support multiple players by addressing the higher demand on network bandwidth and latency.

Applicability to non-VR apps. Finally, the local-remote cooperative rendering architecture of FURION can be applied to non-VR mobile apps such as 3D games which similarly contain foreground interactions and continuously changing background environment to enhance their responsiveness and visual quality and reduce their battery drain.

Pre-rendering versus Real-Time Rendering. As we have shown in Section 3, rendering and streaming encoded VR content in real-time involves unacceptable latency on existing commodity smartphones and wireless networks. FURION pre-renders panoramic frames of background environment on the server to reduce the end-to-end latency. Recently, Liu et al. proposed a novel approach Parallel Rendering and Streaming Pipeline (PRS) [34] that can reduce the real-time VR rendering and streaming latency to be under 14 ms over wireless networks. However, PRS is implemented and evaluated on more powerful hardware as compared to FURION. Specifically, PRS system achieves low end-to-end latency on a laptop equipped with Intel i7-7600U CPU together with dedicated hardware decoder, which is much more powerful than commodity smartphone, and PRS system streams data via 60 GHz WiFi which also has higher throughput than commodity WiFi. PRS is complementary to FURION, and we will study on the combination of PRS and FURION in the future.

11 RELATED WORK

In this section we discuss closely related work in several areas.

Mobile Virtual Reality. Several recent work studied understanding and improving the performance on current VR systems. Chang et al. [35] experimentally quantified the timing and positioning accuracy of existing VR head-mounted hardware. Sun et al. [36] proposed a system to match a given pair of virtual and physical worlds for immersive VR navigation. To enable high-quality VR on mobile devices, Flashback [11] conducted a groundbreaking work that pre-renders all possible views for different positions and orientations and stores them on a local cache. At runtime, it fetches frames on-demand from the cache according to current orientation and position. However, Flashback does not support interactions which update animation/behavior based on user actions, and Pre-caching all possible views locally incurs overwhelming storage overhead (e.g., 50 GB for one app) on the phone. MoVR

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

[13], [14] tries to cut the cord in high-quality VR by using mmWave technology to enable multi-Gbps wireless communication between VR headsets and the server, but it relies on specialized hardware support not present in current VR headsets or smartphones. PIA [37] is a robust 60 GHz network architecture that can provide seamless coverage and mobility support at multi-Gbps bitrate for emerging mobile apps such as VR/AR. More recently, Liu et al. [34] proposed Parallel Rendering and Streaming Pipeline and Remote VSync Driven Frame Rendering (RVDR) to jointly optimize the streaming and display latency of mobile VR systems. Their work is complementary to FURION and can be integrated into FURION to further reduce the latency. In early 2017, Qualcomm introduced the newest premium-tier mobile platform Snapdragon 835, together with a new virtual reality development kit (VRDK) [38] which is designed to meet the demanding requirements of mobile VR. Such hardware enhancement complements our work.

Cloud Offloading and Cloud Gaming. Since the VR system has imposed significant computation overhead on the mobile device, a straightforward approach to improving VR performance is to offload the rendering task to a remote server. The mobile computing community has a long history of leveraging cloud offloading to supplement the capabilities of resource-constrained mobile handsets (e.g., [39], [40], [41], [42], [43], [44]). For example, CloneCloud [40] and MAUI [39] leverage features of managed-language runtimes to automatically partition a program depending on runtime prediction of the partition's energy and performance benefits. However, these works focus on general computation workloads and cannot meet the strict latency requirement specific to VR systems. Similarly, several recent studies focus on real-time rendering/cloud gaming for mobile devices [30], [45], [46], [47], [48], [49], [50], [51]. Again, the latency requirement of VR systems (e.g., under 25 ms) is much more stringent than computer games (e.g., 100-200 ms [46]) due to the near-eye setting.

Graphics and Video Processing. The graphics and video community has also studied efficient rendering on resource-constrained devices. For example, Image-Based Rendering (IBR) [16], [52] is a well studied technology that renders new frames based on a previous image and extra information (e.g., a new angle). However, IBR-like methods do not support well dynamic objects (e.g., interactions that change due to user actions) which may lead to visual artifacts. Parallel codec for traditional video playback has been studied in recent years [53], [54], [55]. FURION makes a special optimization by modifying the codec scheme to support direct encoding/decoding. In addition, several previous studies leverage the viewpoint-adaptive mechanism [56], [57], [58], [59], [60], which reduces the resolution for the pixels outside the field of view to cut down the bitrate of the stream. Recent studies like Rubiks [61] and Flare [62] leverage the adaptive tile-based layered video encoding scheme to stream 360-degree videos for smartphones. However, panoramic videos are not interactive and the FPS requirement of videos is often looser than interactive VR apps. Similarly, foveated rendering leverages eye-tracking to cut down resolution outside the eye gaze [63], [64], [65]. These approaches do not support user interactions and do not perform well when the user quickly and randomly turns the headset and changes the FoV.

12 CONCLUSION

We presented a quantitative, performance-driven design study that shows the QoE achievable for high-quality VR apps on today's mobile hardware and wireless networking is about 10X away from the acceptable QoE and waiting for future mobile hardware or next-generation wireless networks (e.g., 5G) is unlikely to help.

Guided by the quantitative, performance-driven design study, we developed FURION, a VR framework that enables high-quality, immersive mobile VR on today's mobile devices and wireless networks. FURION employs a split renderer architecture running on both the phone and the server to exploit a key insight about the high-quality VR workload that foreground interactions and background environment have contrasting predictability and rendering workload. We implemented FURION on top of Unity and Google Daydream and our evaluation shows it can support high-quality VR apps on today's smartphones over WiFi, with 60 FPS and under 14 ms, 1 ms, and 12 ms responsiveness to controller, rotation, and movement interactions, respectively.

ACKNOWLEDGMENTS

This project is supported by the National Key R&D Program of China under Grant 2017YFB1010002, NSF CCF-1320764 and NSFC (No. 61872211).

REFERENCES

[1] Vr and augmented reality will soon be worth $150 billion. here are the major players. [Online]. Available: https://www.fastcompany.com/3052209/tech-forecast/vr%2Dand%2Daugmented%2Dreality%2Dwill%2Dsoon%2Dbe%2Dworth%2D150%2Dbillion%2Dhere%2Dare% 2Dthe%2Dmajor%2Dpla, Last accessed 2017.

[2] Oculus rift. [Online]. Available: https://www3.oculus.com/en-us/rift/, Last accessed 2017.

[3] Htc vive. [Online]. Available: https://www.vive.com/us/, Last accessed 2017.

[4] Daydream-ready smartphones. [Online]. Available: https://vr.google.com/daydream/phones/, Last accessed 2017.

[5] 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 *Proc. 23rd Annu. Int. Conf. Mobile Comput. Netw.*, 2017, pp. 409–421.

[6] Unity 3d. [Online]. Available: https://unity3d.com, Last accessed 2017.

[7] Google daydream. [Online]. Available: https://vr.google.com/daydream/, Last accessed 2017.

[8] Viking village. [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/29140, Last accessed 2017.

[9] Corridor. [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/33630, Last accessed 2017.

[10] Nature. [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/52977, Last accessed 2017.

[11] K. Boos, D. Chu, and E. Cuervo, "Flashback: Immersive virtual reality on mobile devices via rendering memoization," in *Proc. 14th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2016, pp. 291–304.

[12] What vr could, should, and almost certainly will be within two years. [Online]. Available: http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%20 2014.pdf, Last accessed 2014.

[13] O. Abari, D. Bharadia, A. Duffield, and D. Katabi, "Cutting the cord in virtual reality," in *Proc. 15th ACM Workshop Hot Top. Netw.*, 2016, pp. 162–168.

[14] O. Abari, D. Bharadia, A. Duffield, and D. Katabi, "Enabling high-quality untethered virtual reality," in *Proc. 14th USENIX Conf. Networked Syst. Des. Implementation*, 2017, pp. 531–544.

[15] Gear vr. [Online]. Available: http://www.samsung.com/global/galaxy/gear-vr/, Last accessed 2017.

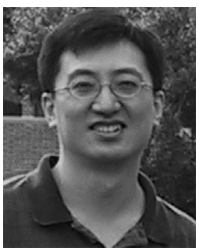
Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.

[16] B. Reinert, J. Kopf, T. Ritschel, E. Cuervo, D. Chu, and H.-P. Seidel, "Proxy-guided image-based rendering for mobile devices," *Comput. Graph. Forum*, vol. 35, no. 7, pp. 353–362, 2016.
[17] Polyrunner vr. [Online]. Available: https://play.google.com/store/apps/details?id=com.lucidsight.polyrunner vr&hl=en
[18] Lego. [Online]. Available: https://play.google.com/store/apps/details?id=com.lego.brickheadz.dream builder&hl=en
[19] vtime. [Online]. Available: https://play.google.com/store/apps/details?id=net.vtime.cardboard&hl=en
[20] Overlord. [Online]. Available: https://play.google.com/store/apps/details?id=com.otherside.underworldo verlord&hl=en
[21] A. Mittal, A. K. Moorthy, and A. C. Bovik, "No-reference image quality assessment in the spatial domain," *IEEE Trans. Image Process.*, vol. 21, no. 12, pp. 4695–4708, Dec. 2012.
[22] W. Li, D. Wu, R. K. Chang, and R. K. Mok, "Demystifying and puncturing the inflated delay in smartphone-based wifi network measurement," in *Proc. 12th Int. Conf. Emerging Netw. EXperiments Technol.*, 2016, pp. 497–504.
[23] x264library. [Online]. Available: http://www.videolan.org/developers/x264.html, Last accessed 2017.
[24] Vp9. [Online]. Available: https://www.webmproject.org/vp9/, Last accessed 2017.
[25] mjpeg tool. [Online]. Available: http://mjpeg.sourceforge.net/, Last accessed 2017.
[26] libx-vp9. [Online]. Available: https://trac.ffmpeg.org/wiki/Encode/VP9, Last accessed 2017.
[27] Android mediacodec. [Online]. Available: https://developer.android.com/reference/android/media/MediaCodec.html, Last accessed 2018.
[28] ffmpeg. [Online]. Available: https://ffmpeg.org/, Last accessed 2017.
[29] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, Apr. 2004.
[30] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi, "Kahawai: High-quality mobile gaming using gpu offload," in *Proc. 13th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2015, pp. 121–135.
[31] Thermal requirement. [Online]. Available: https://developers.google.com/vr/distribute/daydream/performance-requirements, Last accessed 2017.
[32] Battery historian tool. [Online]. Available: https://github.com/google/battery-historian, Last accessed 2018.
[33] Altspace vr. [Online]. Available: https://play.google.com/store/apps/details?id=com.altvr.AltspaceVR&hl=en, Last accessed 2018.
[34] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser, "Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering," in *Proc. 16th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2018, pp. 68–80.
[35] C.-M. Chang, C.-H. Hsu, C.-F. Hsu, and K.-T. Chen, "Performance measurements of virtual reality systems: Quantifying the timing and positioning accuracy," in *Proc. ACM Multimedia Conf.*, 2016, pp. 655–659.
[36] Q. Sun, L.-Y. Wei, and A. Kaufman, "Mapping virtual and physical reality," *ACM Trans. Graph.*, vol. 35, no. 4, 2016, Art. no. 64.
[37] T. Wei and X. Zhang, "Pose information assisted 60 ghz networks: Towards seamless coverage and mobility support," in *Proc. 23rd Annu. Int. Conf. Mobile Comput. Netw.*, 2017, pp. 42–55.
[38] Qualcomm snapdragon 835. [Online]. Available: https://www.qualcomm.com/news/releases/2017/02/23/qualcomm%2Dintroduces%2Dsnapdragon%2D835%2Dvirtual%2Dreality%2Ddevelopment%2Dkit, Last accessed 2017.
[39] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Serv.*, 2010, pp. 49–62.
[40] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.
[41] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 93–106.
[42] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2014, pp. 68–81.
[43] D. Narayanan and M. Satyanarayanan, "Predictive resource management for wearable computing," in *Proc. 1st Int. Conf. Mobile Syst. Appl. Serv.*, 2003, pp. 113–128.
[44] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," in *Proc. ACM Symp. Operating Syst. Principles*, 1997, pp. 276–287.
[45] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in *Proc. 13th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2015, pp. 151–165.
[46] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: Architecture and performance," *IEEE Netw.*, vol. 27, no. 4, pp. 16–21, Jul./Aug. 2013.
[47] R. Shea and J. Liu, "On gpu pass-through performance for cloud gaming: Experiments and analysis," in *Proc. Annu. Workshop Netw. Syst. Support Games*, 2013, pp. 1–6.
[48] R. Shea, D. Fu, and J. Liu, "Rhizome: Utilizing the public cloud to provide 3d gaming infrastructure," in *Proc. 6th ACM Multimedia Syst. Conf.*, 2015, pp. 97–100.
[49] S. Shi and C.-H. Hsu, "A survey of interactive remote rendering systems," *ACM Comput. Surveys*, vol. 47, no. 4, 2015, Art. no. 57.
[50] J. Wu, C. Yuen, N.-M. Cheung, J. Chen, and C. W. Chen, "Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 12, pp. 1988–2001, Dec. 2015.
[51] S. Shi, C.-H. Hsu, K. Nahrstedt, and R. Campbell, "Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming," in *Proc. 19th ACM Int. Conf. Multimedia*, 2011, pp. 103–112.
[52] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in *Proc. Symp. Interactive 3D Graph.*, 1997, pp. 7–ff.
[53] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer, "Efficient parallelization of h. 264 decoding with macro block level scheduling," in *Proc. IEEE Int. Conf. Multimedia Expo*, 2007, pp. 1874–1877.
[54] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, "Parallel scalability and efficiency of hevc parallelization approaches," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1827–1838, Dec. 2012.
[55] M. Alvarez-Mesa, C. C. Chi, B. Juurlink, V. George, and T. Schierl, "Parallel video decoding in the emerging hevc standard," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process*, 2012, pp. 1545–1548.
[56] Next generation video encoding for 360 video. [Online]. Available: https://code.facebook.com/posts/1126354007399553/next%2Dgeneration%2D video%2Dencoding%2Dtechniques%2Dfor%2D360%2Dvideo%2Dand%2Dvr/, Last accessed 2017.
[57] F. Qian, L. Ji, B. Han, and V. Gopalakrishnan, "Optimizing 360 video delivery over cellular networks," in *Proc. 5th Workshop All Things Cellular: Operations, Appl. Challenges*, 2016, pp. 1–6.
[58] K. K. Sreedhar, A. Aminlou, M. M. Hannuksela, and M. Gabbouj, "Viewport-adaptive encoding and streaming of 360-degree video for virtual reality applications," in *Proc. Int. Symp. Multimedia*, 2016, pp. 583–586.
[59] D. Ochi, Y. Kunita, K. Fujii, A. Kojima, S. Iwaki, and J. Hirose, "Hmd viewing spherical video streaming system," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 763–764.
[60] V. R. Gaddam, M. Riegler, R. Eg, C. Griwodz, and P. Halvorsen, "Tiling in interactive panoramic video: Approaches and evaluation," *IEEE Trans. Multimedia*, vol. 18, no. 9, pp. 1819–1831, Sep. 2016.
[61] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han, "Rubiks: Practical 360-degree video streaming for smartphones," in *Proc. 16th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2018, pp. 482–494.
[62] F. Qian, B. Han, Q. Xiao, and V. Gopalakrishnan, "Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices," in *Proc. 24th Annu. Int. Conf. Mobile Comput. Netw.*, 2018, pp. 99–114.
[63] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder, "Foveated 3d graphics," *ACM Trans. Graph.*, vol. 31, no. 6, 2012, Art. no. 164.
[64] N. T. Swafford, J. A. Iglesias-Guitian, C. Koniaris, B. Moon, D. Cosker, and K. Mitchell, "User, metric, and computational evaluation of foveated rendering methods," in *Proc. ACM Symp. Appl. Perception*, 2016, pp. 7–14.
[65] L. Zhang, X.-Y. Li, W. Huang, K. Liu, S. Zong, X. Jian, P. Feng, T. Jung, and Y. Liu, "It starts with igaze: Visual attention driven networking with smart glasses," in *Proc. 20th Annu. Int. Conf. Mobile Comput. Netw.*, 2014, pp. 91–102.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.



Zeqi Lai received the BE degree from the School of Computer Science and Engineering, University of Electronic Science and Technology of China. He is now working toward the PhD degree in the Department of Computer Science and Technology, Tsinghua University, working with Prof. Yong Cui. His research interests include mobile systems, cloud computing and cloud storage, and virtual reality.



Y. Charlie Hu received the PhD degree in computer science from Harvard University, in 1997. He is a professor of electrical and computer engineering with Purdue University. From 1997 to 2001, he was a research scientist with Rice University. His research interests include operating systems, distributed systems, Internet measurement and routing analysis, and wireless networking. He has published more than 130 papers in these areas. He received the NSF CAREER Award in 2003. He is a fellow of the IEEE and an ACM distinguished scientist.



Yong Cui received the BE and PhD degrees in computer science and engineering from Tsinghua University, Beijing, China, in 1999 and 2004, respectively. He is currently a full professor with Tsinghua University and co-chair of IETF IPv6 Transition WG Softwire. His major research interests include mobile wireless Internet and computer network architecture. He has published more than 100 papers in refereed journals and conferences, holds more than 40 patents, and authored three Internet standard documents, including RFC 7040 and RFC 5565, for his proposal on IPv6 transition technologies. He is a member of the IEEE.



Linhui Sun received the BE degree in telecommunications engineering with management from the Beijing University of Posts and Telecommunications, in 2015. He is now working toward the master's degree in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, advised by Prof. Yong Cui. His research interests include network protocols and virtual reality.



Ningwei Dai received the BEng degree in communication engineering from the Beijing University of Posts and Telecommunications, the Beijing, China, in 2015. She is currently working towards the master's degree in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Her research interests include networking and cloud computing.



Hung-Sheng Lee received the BE degree in software engineering from BeiHang University, in 2017. He is now working toward the MS degree in the Department of Computer Science and Technology, Tsinghua University, advised by Prof. Yong Cui. His research interest is broadly in networked systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 15,2022 at 03:34:28 UTC from IEEE Xplore. Restrictions apply.