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

Zeqi Lai*
Tsinghua University
laizq13@mails.tsinghua.edu.cn

Y. Charlie Hu
Purdue University
ychu@purdue.edu

Yong Cui
Tsinghua University
cuiyong@tsinghua.edu.cn

Linhui Sun
Tsinghua University
lh.sunlinh@gmail.com

Ningwei Dai
Tsinghua University
lemondnw@gmail.com

ABSTRACT

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, and parallel decoding on multiple cores on the phone, we demonstrate Furion can support high-quality VR apps on today's smartphones over WiFi, with under 14ms latency and 60 FPS (the phone display refresh rate).

KEYWORDS

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 [24].

Despite the growing market penetration, today's high-end VR systems such as Oculus Rift [16] and HTC Vive [9] 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" [5], 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 63-111ms and exhausts the phone CPU and GPU with utilization at about 60% and 100%, while on-demand streaming a frame rendered at the server over 802.11ac takes about 230ms.

Second, given neither local rendering nor remote rendering can support untethered VR on today's mobile hardware and wireless networks, we ask the question – will waiting for tomorrow's mobile hardware and next-generation wireless networks help?

Through analyzing the technology trend of mobile hardware and wireless networks we show that waiting for future mobile hardware or next-generation wireless networks (*e.g.,* 5G, 802.11ad) is unlikely to help. First, the CPU/GPU of mobile handsets will not grow significantly more powerful because of the power constraint. In fact, the technology trend shows the CPU capability of high-end phones has largely converged to using quad-cores at 2-2.5 GHz, with aggressive temperature control and hence frequency capping. Second, the multi-Gbps bandwidth promised by the next generation of wireless networks will not make remote rendering readily feasible, because packet processing at 10X higher data rate

*Work done while a visiting student at Purdue University.

MobiCom'17, October 16–20, 2017, Snowbird, UT, USA.

ACM ISBN 978-1-4503-4916-1/17/10…$15.00
https://doi.org/10.1145/3117811.3117815

will far exceed the mobile CPU capability, which as just discussed is unlikely to improve significantly. We show packet processing of 4 Gbps TCP flows would require 16 equivalent cores on Pixel running at the capped frequency.

The above observation and reasoning suggest that the likely path to support untethered high-quality VR, *i.e.,* meeting its QoE requirements, is via software innovation. Furthermore, if such a software solution exists, following the same reasoning above, it should work on today's mobile hardware and wireless networks. In the third 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 32ms to get the frame and 16ms to decode it on the phone, for a total of 48ms end-to-end delay, which is about 3X beyond the 16ms 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 prefetching 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 16ms, 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.

We have implemented the complete Furion platform on top of Unity [21] and Google Daydream [8]. 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 [22], Corridor [4] and Nature [13]. We show Furion can support high-quality VR apps on today's smartphone (*e.g.,*Pixel) over WiFi (802.11ac), by providing under 14ms 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 dynamic objects 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% CPU utilization), which allows it to sustain long running of VR apps without being restricted by temperature control and frequency capping. Finally, by moving on-demand fetching of frames from the server out of the end-to-end delay critical path, *i.e.,* turning it into prefetching which just needs to finish in the prefetching window, Furion is robust to transient network bandwidth fluctuations. 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.

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.

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 [31]. 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:

(1) **Responsiveness:** The motion-to-photon latency should be low enough, *e.g.,* under 10-25ms [26], 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 [31]).
(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 [28, 29].

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

Figure 1: A subset of VR apps used in our study: PolyRunner is low-quality, and the rest are high-quality.

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.*, OculusRift [16], HTC Vive [9]) 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 [8] and Samsung Gear VR [7] 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 [31, 51], 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 [18], Lego [10], vTime [25] and Overlord [17] are popular but low-quality Android VR apps available in Google Play which all have controller interactions, and Viking Village [22], Corridor [4] and Nature [13] are high-quality virtual-world Unity apps with more complex, realistic and immersive environments as shown in Figure 1. 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.

We ran the seven VR apps on a Pixel smartphone running Android 7.1. We used FPS and the BRISQUE value [46] 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.[1]

Table 1: QoE and CPU/GPU load when running different VR apps on Pixel. A lower BRISQUE value of an app indicates better visual quality.

App Name	BRISQUE[46] Score	FPS	CPU Load	GPU Load
Low-quality VR apps				
PolyRunner [18]	89.19	60	36.2%	68.9%
Lego [10]	59.31	52	39.3%	74.4%
vTime [25]	54.48	47	41.7%	70.3%
Overlord [17]	50.10	41	44.8%	84.9%
High-quality VR apps				
Viking [22]	24.37	11	55.8%	99.9%
Corridor [4]	26.67	9	55.5%	99.8%
Nature [13]	26.67	16	59.9%	99.8%

The results in Table 1 show that (1) even the first four, low-quality VR apps incur up to 85% GPU utilization, while driving the CPU at 40% utilization[2] 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% utilization) and CPU (55-59% 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 63-111ms which is far longer than the required 16ms per-frame rendering interval.

To summarize, under local rendering, the time to render a frame (roughly the inverse of each app's FPS) is:

$$T_{phone\_render} = 63 \sim 111ms \quad (1)$$

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 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. Under remote rendering, the time to render a frame is:

$$T_{remote\_render} = T_{req} + T_{server\_render} + T_{transfer} \quad (2)$$

[1] There is no well-defined threshold BRISQUE value for defining acceptable image qualities.

[2] Throughout the paper, phone CPU utilization refers to the average utilization of all cores.

Table 2: Latency breakdown when we stream high-quality VR content from a strong desktop to a smartphone via 802.11ac WLAN. The average FPS of each app is about 5. The peak network throughput is about 400Mbps.

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

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 256GB 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. 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. To support 60 FPS or higher requires streaming each high-quality frame well under 16ms, 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). A recent study [28] also confirmed multi-Gbps bandwidth is needed for supporting high-quality VR apps in remote rendering.

(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 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 [44]). Figure 2 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 2ms minimum to 4ms at the 80th percentile, with a long tail reaching beyond 6ms. These results are consistent with those observed in [44]. The RTT can be explained by about 2ms base latency of PHY layer transfer (RTS/CTS/DATA/ACK) and the 1ms (median) delay in traversing the hardware/driver/kernel stack on the phone, which is again limited by the phone hardware capability.

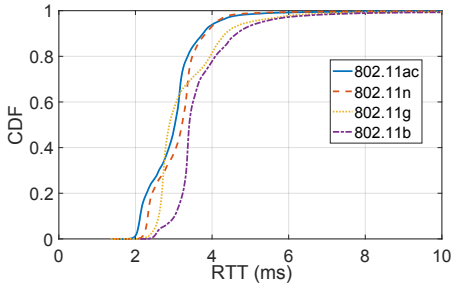


Figure 2: App-level RTT in different WiFi networks.

In summary, on today's high-end handset and highest bandwidth wireless network 802.11ac, the time to render a frame under on-demand remote rendering is:

$$\underset{(230ms)}{T_{remote\_render}} = \underset{(3ms)}{T_{req}} + \underset{(11ms)}{T_{server\_render}} + \underset{(216ms)}{T_{transfer}} \quad (3)$$

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.

3 WILL FUTURE HARDWARE/NETWORKS HELP?

Given neither local rendering nor remote rendering can support untethered VR on today's mobile hardware and wireless networks, we ask the question – will waiting for tomorrow's mobile hardware and wireless networks help?

Mobile hardware are power limited. We argue that *the CPU/GPU of mobile handsets will not grow significantly more powerful because of the power constraint.* First, just like their counterpart in desktops and servers, the CPUs in mobile handsets are not getting faster, due to temperature and power limitation in the chip design. Second, since mobile handsets are fundamentally energy constrained, the CPUs on mobile devices are made to be less powerful (though often more power efficient), and come with far fewer cores, than their counterparts on desktops and servers.

In fact, the improvement of CPU/GPU capability of mobile handsets in high-end mobile handsets has already tapered off in recent years. Table 3 lists the CPU/GPU frequency and number of cores for the high-end Android handsets in the past five years and the upcoming S8+ (at the time of paper submission). We observe that (1) the CPU in these high-end phones has more or less converged to quad cores; although Nexus 6P comes with Octa cores, the upper four cores are rarely turned on due to temperature control; (2) the CPU frequency appears to have converged to between 2-2.5 GHz; (3) while the GPU frequency appears to be slowly increasing, we believe its provision benefited from larger batteries that come with larger phones adopted by the consumers in the past few years; the phone size clearly will not forever increase.

The stagnating improvement of CPU/GPU capabilities in mobile handsets suggests that local rendering will unlikely become a viable solution to support untethered VR apps.

Table 3: Evolution of CPU/GPU in high-end mobile handsets. Higher frequency benefited from larger batteries.

Phone (year)	Battery (size)	CPU	GPU
Nexus 4 (2012)	2100 mAh (4.7 inches)	Quad-core 1.5 GHz Krait	Adreno 320 400 MHz
Nexus 5 (2013)	2300 mAh (5.0 inches)	Quad-core 2.3 GHz Krait	Adreno 330 450 MHz
Nexus 6 (2014)	3220 mAh (6.0 inches)	Quad-core 2.7 GHz Krait	Adreno 420 500 MHz
Nexus 6P (2015)	3450 mAh (5.7 inches)	Octa-core 2.0+1.55 GHz Cortex-A57+A53	Adreno 430 600 MHz
Pixel XL (2016)	3450 mAh (5.5 inches)	Quad-core 2.15+1.6 GHz Kryo (2 sm, 2 lg)	Adreno 530 650 MHz
Galaxy S8+ (2017)	3500 mAh (6.2 inches)	Octa-core 2.45 GHz Kryo	Adreno 540 710 MHz

Table 4: Evolution of wireless technologies. We plot both theoretical peak bandwidth and achieved peak throughput of different cellular and WiFi networks.

Type	EDGE	UMTS	HSPA	LTE
Theoretical	1 Mbps	7.2 Mbps	42 Mbps	100 Mbps
Practical	384 Kbps	2 Mbps	10 Mbps	20 Mbps

Type	802.11b	802.11g	802.11n	802.11ac
Theoretical	11 Mbps	54 Mbps	600 Mbps	1.3 Gbps
Practical	6 Mbps	20 Mbps	100 Mbps	400 Mbps

Will next-generation wireless networks help? Compared to mobile CPU/GPU evolution, WiFi and cellular networking technologies have seen continuous and more rapid improvement. Table 4 lists the peak bandwidth provided by the generations of WiFi and Cellular. We see that in the past 10 years, the deployed cellular networks have evolved from 2.5G to 3G to LTE/4G, with achieved peak throughput growing from 384Kbps in 2.5G to 20 Mbps in LTE. Development of 5G started in year 2012 and wide deployment is estimated to happen in 2020. 5G is promised to deliver up to 1Gbps peak bandwidth [15], or 10X that of LTE/4G. Similarly, WiFi has evolved from 802.11a/b to 802.11ac which has a theoretical peak bandwidth of 1.3 Gbps and delivers up to 400 Mbps TCP throughput in practice, with 802.11ad promising a theoretical peak bandwidth of 7 Gbps.

However, we argue *the multi-Gbps bandwidth promised by the next-generation wireless networks will not make remote rendering readily feasible*, for the following reason.

Packet processing will exhaust the mobile CPU.

Packet processing at high throughput, which includes interrupt-driven packet receiving and stack processing, consumes a significant amount of CPU [39]. Since mobile CPUs will not become much faster or employ many more cores as discussed above, they will not be able to keep up with the 10X throughput increase.

To quantify how much the CPU is already consumed by packet processing, we measured the CPU utilization of Pixel when running a microbenchmark that performs downloading from a server connected via an 802.11ac WiFi AP under TCP. We ran the Linux tool `tc` on the server to control the peak throughput between the phone and server.

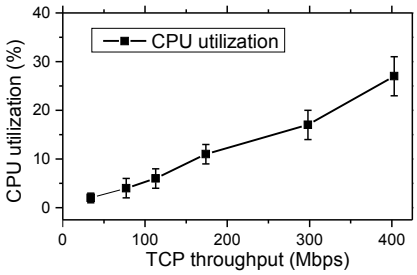


Figure 3: CPU utilization of Pixel from packet processing under varying downloading throughput.

Figure 3 plots the CPU utilization as the TCP throughput varies. We observe that (1) at 400 Mbps, the average utilization of the 4 cores of Pixel is already at 27%; (2) the CPU utilization scales roughly linearly with the TCP throughput; (3) under linear extrapolation, the CPU utilization would reach an unrealistic 270% at 4 Gbps (or 70% on 16 equivalent cores of Pixel).

Need software innovation. Given that waiting for the future mobile hardware or next-generation wireless networks is unlikely to 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.

4 EXPLORING VIDEO COMPRESSION

We start our quest for a software solution to support untethered VR apps today with the remote rendering approach discussed in §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. This is achieved by encoding the reference frame into an I frame which is relatively large and self-contained, and encoding the subsequent frames as much smaller, dependent P frames which encode the delta relative to the reference I frame and proceeding P frames.

Recall the original remote rendering approach suffers about 200ms 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 [27] and VP9 [23], 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 [12], libx264 [27] or libx-vp9 [11] to encode the frame on the server and leverage Android MediaCodec [2] to decode frames on the phone.

We then reran the three high-end VR apps on Pixel and measured the QoE when the server streams compressed frames in MPEG, H.264 or VP9. Table 5 shows that (1) MJPG compression incurs an encoding time of up to 17ms and reduces the average frame size to about 400KB for the three VR apps which cuts down the frame transfer time to 14ms, but it also incurs a high decoding delay of 128-137ms on the phone, which makes the end-to-end delay in remote rendering a new frame on-demand stay at about 179ms (3ms + 11ms + 17ms + 11ms + 137ms). (2) Although VP9 achieves the highest compression ratio and smallest per-frame size,

Table 5: Latency breakdown when we stream high-quality VR content from a strong desktop to a smartphone via 802.11ac WLAN, with video compression. M:MJPG. H:H.264. V:VP9. In each case the peak network throughput is about 400Mbps.

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

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 10ms and 8ms, it also cuts down the phone decoding time to be under 16ms. 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:

$$\underset{(48ms)}{T_{remote\_render}} = \underset{(3ms)}{T_{req}} + \underset{(11ms)}{T_{render}} + \underset{(10ms)}{T_{encode}} + \underset{(8ms)}{T_{transfer}} + \underset{(16ms)}{T_{phone\_decode}} \quad (4)$$

Examining each of the five components of the end-to-end delay of 48ms, 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 §3. Therefore, although the end-to-end delay of 48ms is only 3X from the per-frame interval of 16ms, the gap appears very challenging to bridge.

5 KEY INSIGHT

The five components of the end-to-end time to render a frame in Equation (4) can be separated into two groups. The first group, consisting of the first four components, includes the delay in on-demand requesting, server rendering and encoding the new frame, and transferring the new frame over the network. The second group consists of the time to decode the compressed frame on the phone.

To gain insight on how to reduce the delay in the first group, we studied more than 50 popular VR apps collected from Google Daydream platform and Unity Store to understand the VR rendering workload. From the study, we derived four VR-specific insights:

(1) For most VR apps, the VR content rendered can be divided into *foreground interactions* and *background virtual environment*. For example, for the VR app we have tested in Figure 1, the interaction is the flyer in Figure 1a, the axe in Figure 1b, the gun or enemies in Figure 1c, or the animals in Figure 1d, which interact with the user via the controller. The environment refers to the background virtual world that covers the majority of the space in the display.
(2) Interactions are triggered by operating the controller or signals from other players. As a result, their animations are random and hard to predict.
(3) In contrast, the background environment is updated according to the user movement and thus changes continuously and is predictable. However, at a given location, the user may change orientation abruptly, *e.g.,* from turning, and the environment seen from the new viewing angle may change significantly, although the environment itself did not change.

Table 6: Resource usage when rendering only foreground interactions or background environment. TPF is average rendering time per frame.

App	Interaction only (TPF/CPU/GPU)	Environment only (TPF/CPU/GPU)
Viking	13ms / 29% / 22%	82ms / 42% / 95%
Corridors	13ms / 40% / 22%	104ms / 53% / 83%
Nature	12ms / 32% / 20%	61ms / 56% / 82%

Next, we break down the cost of rendering foreground interactions and background environment for the three high-quality VR apps in Figure 1 on Pixel. Specifically, for each app we build two new versions that disable the call for drawing foreground interactions and the background environment, respectively. We then measure the frame rendering time and CPU and GPU utilization of each version. Table 6 shows the results. We see that rendering interactions only takes 12-13ms, which is 7X shorter than rendering both interactions and environment (§2.3.1), and consumes only 29-40% CPU utilization and under 22% GPU utilization on the Pixel phone. In contrast, rendering only the environment takes 61-104ms which is about 90% of the delay in rendering both interactions and environment (§2.3.1). Thus we draw our next insight about VR rendering workload:

(4) In terms of rendering workload, foreground interactions are much more lightweight compared to the background environment. This is because the realistic, immersive environment in high-quality VR apps contains rich details and complex textures (*e.g.,* the sophisticated buildings and shadows in Figure 1) which incur significant rendering overhead.

6 COOPERATIVE RENDERING

6.1 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 the background environment, and finally,
(4) combine them on the phone to generate the final frames.

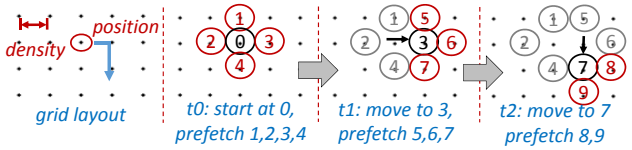


Figure 4: The prefetching process.

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:

$$T_{co\_render} = max(T_{phone\_render\_intr}, T_{phone\_decode\_env}) + T_{integrate} \quad (5)$$

Constraint: environment frames are pre-fetched in time

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.

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 16ms:

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 16ms interval?

6.2 Prefetching panoramic frames

In the basic cooperative renderer design, we assume the next frame to be rendered can be pre-rendered and prefetched. 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).

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 Figure 4, and rendering the faraway environment at any location in between grid points is approximated with rendering from the nearest grid point. We define the distance between two adjacent positions as *density (d)*. Effectively the grid density determines the smoothness of rendering during user movement in the virtual world. Previous studies (*e.g.,* [31]) have shown that for high-end VR apps, setting the density to 0.02 Unity units (which corresponds to about 2cm) is sufficient to sustain smooth rendering during user movement.

Table 7: 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	≤ 3 panoramas (3*31 MB)
Panorama+ +Video compression	1 video (300-400 KB)

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 20ms 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 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 Figure 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. 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*10MB down to 3*31MB when without video compression, as shown in Table 7. With video compression,

Table 8: Latency of each step when we prefetch the background environment for Viking Village. Latencies of other two VR apps are similar.

On-demand	$T_{phone\_render\_intr}$	T_{phone_decode}	
	13ms	45ms	
Prefetching	T_{req}	$T_{transfer}$	T_{walk}
	3ms	10ms	20ms

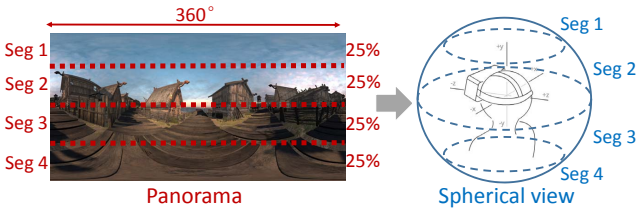


Figure 5: Parallel decoding: leveraging multi-core to decode frame segments in parallel.

one panoramic frame can be compressed down to a P-frame of under 133KB on average (using the panoramic frame at the current grid point as reference).

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 110ms, as decoding the last of the three panoramic frame in the video requires decoding the first two. For this reason, we encode each of the three panoramic frames (*e.g.,* for position 5, 6, and 7) as an independent P frame relative to the reference frame that the phone already has (*e.g.,* for position 3). Making the three P-frames not dependent this way reduces the decoding time of any P-frame (panoramic frame) to 45ms on average, at the cost of a slightly decreased compression ratio (about 9% reduction). We denote this non-successive variation of video encoding as *direct encoding*.

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 8 shows five components of the process. On one hand, sending the request for and transferring of three panoramic frames can be finished in 13ms, which is less than the 20ms delay for the user to walk from one grid location to the next, and hence satisfying the prefetching constraint in Equation (5):

$$Constraint: \underset{(3ms)}{T_{req}} + \underset{(10ms)}{T_{transfer}} \leq \underset{(20ms)}{T_{walk\_to\_next\_grid\_point}} \quad (6)$$

On the other hand, decoding any one of the three panoramic frames in the compressed video takes about 45ms, much higher than the 16ms for decoding a compressed regular frame (Table 5).

6.3 Parallel decoding on the phone

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

Table 9: Latency of each step with parallel decoding for Viking.

On-demand	$T_{phone\_render\_intr}$	T_{phone_decode}
	13ms	12ms

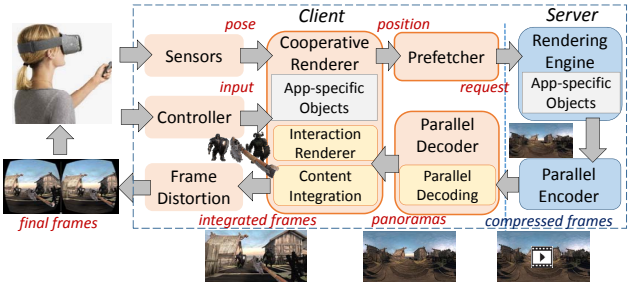


Figure 6: Furion system architecture.

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. Figure 5 shows how a panoramic frame is segmented into 4 segments.

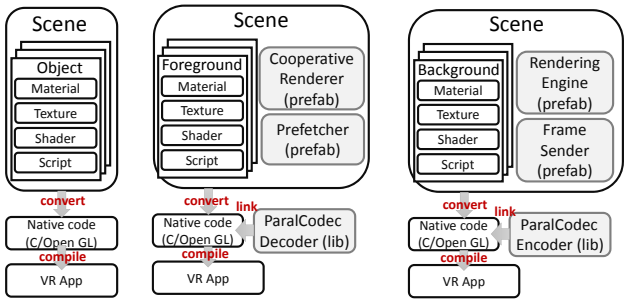
Table 9 shows parallel decoding of four segments reduces the phone decoding time from 45ms to 12ms, and the end-to-end time to render a new frame to $max(13ms, 12ms) + 1ms = 14ms$, which allows the VR apps to achieve 60 FPS.

6.4 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.

Figure 6 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 for panoramic frames corresponding to up to three new adjacent grid locations.

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 up to 3 panoramic frames specified in the client request,



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

Figure 7: Developing VR apps with Furion.

cuts each panoramic frame into 4 segments, encodes them with reference to the panoramic frame (subframes) in the last grid point using H.264 codec, and transfers the encoded video to the phone over the wireless network.

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. Figure 7a 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. Figures 7b and 7c 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.,* 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 [6] and x264 [27] 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.

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 400Mbps TCP throughput for a single device. We augment the three high-quality apps as we described previously (Viking [22], Corridor [4], and Nature [13]) with Furion to support high-quality virtual reality.

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 [28, 29] because there is no 60GHz WiFi support in current commodity smartphones, or with FlashBack [31] which does not support user interactions.

Table 10: 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

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.

9.2 Visual quality, frame rate, responsiveness

Image quality. We use Structural Similarity (SSIM) [59] 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 10 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 [37].

Frame rate. To enable a smooth immersive experience, it is necessary for the VR system to deliver at least 60 FPS. Table 10 shows across the three high-quality VR apps, the mobile system delivers the lowest FPS, between 9-15, the thin-client version delivers between 34-37 FPS, while Furion comfortably delivers 60 FPS. We note that the 60 FPS frame rate of Furion is not limited by our implementation, but by the 60Hz refresh rate of the display hardware on Pixel.

Responsiveness. Following [31], 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.

Table 11: 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

Figure 8 shows Furion achieves far better responsiveness than the other two systems. In particular, it achieves under 14ms controller latency, under 1ms rotation latency, and under 12ms 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.

9.3 Scalability with app features

Next, we evaluate the scalability of different schemes in terms of different app features, in particular, in terms of the number of dynamic objects, 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. Figure 9 shows the result for Viking Village; 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 interactive dynamic objects.

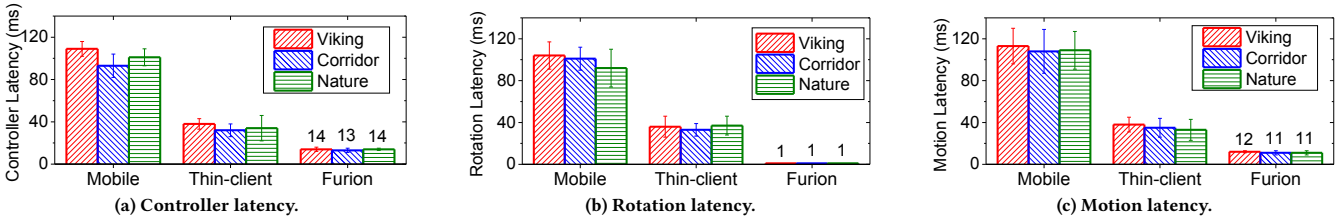
9.4 Resource usage

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

CPU/GPU usage. We first measure the CPU/GPU utilization together with the network usage in Table 11. We see Furion incurs up to 65% CPU utilization and 37% 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 [5], 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 11 shows that running the three apps on Nexus 6P incurs much higher CPU and GPU utilization, up to 76% for CPU and 64% for GPU, suggesting the VR-readiness of Pixel mainly comes from the upgraded GPU (Table 3).

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



(a) Controller latency. (b) Rotation latency. (c) Motion latency.

Figure 8: Responsiveness of different VR implementations.

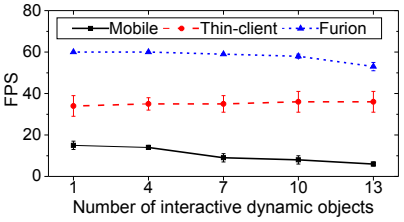


Figure 9: FPS as the number of interactive dynamic objects varies.

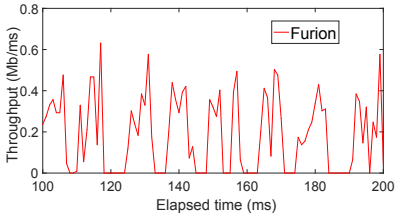


Figure 10: Bandwidth usage in each 20ms window.

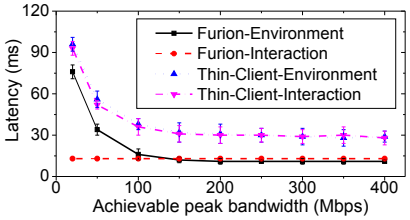


Figure 11: Rendering latency under bandwidth capping.

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 [20] 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. Figure 12(a)(b) 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 [3] to inspect the power drain of the Android device over time. Figure 12(c) 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.

9.5 Robustness to network instability

Table 11 further shows that supporting the three VR apps in Furion requires on average 132 Mbps network throughput, well below the 400Mbps peak throughput supported by 802.11ac. Figure 10 shows this is because prefetching panoramic frames of adjacent positions in the virtual world can quickly complete in a short burst (about 8ms per Equation (4)) within each 20ms interval of moving between two adjacent positions. The throughput value in Figure 10 starts from 100ms 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 20ms window. This property suggests Furion can tolerate transient bandwidth fluctuations as long as the network provides 130Mbps 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. Figure 11 shows under Furion the delay in rendering of interactions and in decoding environment stays constant when the average peak bandwidth is reduced from 400Mbps to 150Mbps.

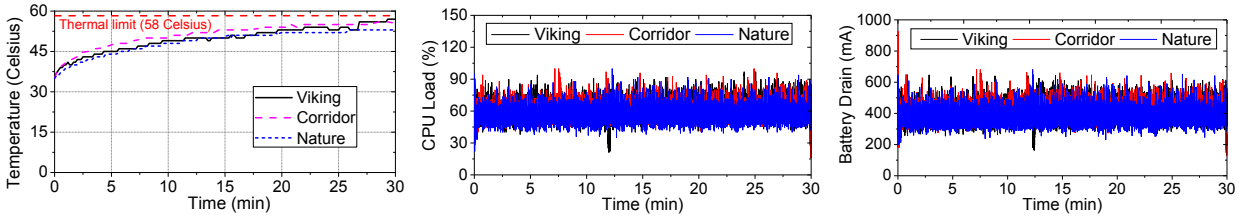
10 DISCUSSIONS

We now discuss how Furion design can be extended to handle more complicated VR app features.

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 6.2, the movement delay is about 20ms 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. 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 points. For example, at moment t1 in Figure 4, if the user moves at twice the normal speed, we just need to prefetch the frames from 2 grid positions away.

Dynamic objects in background environment. For 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 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.

Multi-player support. Some latest VR apps such as vTime [25] and Altspace VR [1] 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.



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

Figure 12: 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.

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.

11 RELATED WORK

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.* [32] experimentally quantified the timing and positioning accuracy of existing VR head-mounted hardware. To enable high-quality VR on mobile devices, Flashback [31] 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. Flashback has two limitations: (1) Since all frames are pre-rendered, it does not support interactions which update animation/behavior based on user actions; (2) Pre-caching all possible views locally incurs overwhelming storage overhead (*e.g.,* 50GB for one app) on the phone. More recently, MoVR [28, 29] 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. In early 2017, Qualcomm introduced the newest premium-tier mobile platform Snapdragon 835, together with a new virtual reality development kit (VRDK) [19] which is designed to meet the demanding requirements of mobile VR and expected to be available in Q2 2017. Such hardware enhancement complements our work.

Cloud offloading and cloud gaming: The mobile computing community has a long history of leveraging cloud offloading to supplement the capabilities of resource-constrained mobile handsets (*e.g.,* [35, 36, 40, 42, 47, 48]). 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 [37, 43, 52–56, 60]. Again, the latency requirement of VR systems (*e.g.,* under 25ms) is much more stringent than computer games (*e.g.,* 100-200ms [54]) 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) [45, 51] is a well studied technology that renders new frames based on a previous image and extra information (*e.g.,* a new angle). More recently, the authors in [51] propose a new IBR algorithm to mask the latency of the rendering task while providing the client with coverage for filling disocclusion holes. 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 [30, 33, 34]. 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 [14, 38, 49, 50, 57], which reduces the resolution for the pixels outside the field of view (FOV) to cut down the bitrate of the stream. Similarly, foveated rendering leverages eye-tracking to cut down resolution outside the eye gaze [41, 58, 61]. These approaches 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 (1) the QoE achievable for high-quality VR apps on today's mobile hardware and wireless networking is about 10X away from the acceptable QoE; (2) 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 14ms, 1ms, and 12ms responsiveness to controller, rotation, and movement interactions, respectively.

13 ACKNOWLEDGEMENTS

We thank our shepherd, Robert LiKamWa, for helpful feedback that greatly improved our paper. We also thank the anonymous reviewers for their valuable comments. This project is supported by NSF CCF-1320764 and NSFC of China (No. 61422206).

REFERENCES

[1] Altspace vr. https://play.google.com/store/apps/details?id=com.altvr.AltspaceVR&hl=en.
[2] Android mediacodec. https://developer.android.com/reference/android/media/MediaCodec.html.
[3] Battery historian tool. https://github.com/google/battery-historian.
[4] Corridor. https://www.assetstore.unity3d.com/en/#!/content/33630.
[5] Daydream-ready smartphones. https://vr.google.com/daydream/phones/.
[6] ffmpeg. https://ffmpeg.org/.
[7] Gear vr. http://www.samsung.com/global/galaxy/gear-vr/.
[8] Google daydream. https://vr.google.com/daydream/.
[9] Htc vive. https://www.vive.com/us/.
[10] Lego. https://play.google.com/store/apps/details?id=com.lego.brickheadz.dreambuilder&hl=en.
[11] libx-vp9. https://trac.ffmpeg.org/wiki/Encode/VP9.
[12] mjpeg tool. http://mjpeg.sourceforge.net/.
[13] Nature. https://www.assetstore.unity3d.com/en/#!/content/52977.
[14] Next generation video encoding for 360 video. https://code.facebook.com/posts/1126354007399553/next%2Dgeneration%2Dvideo%2Dencoding%2Dtechniques%2Dfor%2D360%2Dvideo%2Dand%2Dvr/.
[15] Ngmn 5g white paper. https://www.ngmn.org/uploads/media/NGMN_5G_White_Paper_V1_0.pdf.
[16] Oculus rift. https://www3.oculus.com/en-us/rift/.
[17] Overlord. https://play.google.com/store/apps/details?id=com.otherside.underworldoverlord&hl=en.
[18] Polyrunner vr. https://play.google.com/store/apps/details?id=com.lucidsight.polyrunnervr&hl=en.
[19] Qualcomm snapdragon 835. https://www.qualcomm.com/news/releases/2017/02/23/qualcomm%2Dintroduces%2Dsnapdragon%2D835%2Dvirtual%2Dreality%2Ddevelopment%2Dkit.
[20] Thermal requirement. https://developers.google.com/vr/distribute/daydream/performance-requirements.
[21] Unity 3d. https://unity3d.com.
[22] Viking village. https://www.assetstore.unity3d.com/en/#!/content/29140.
[23] Vp9. https://www.webmproject.org/vp9/.
[24] Vr and augmented reality will soon be worth $150 billion. here are the major players. https://www.fastcompany.com/3052209/tech-forecast/vr%2Dand%2Daugmented%2Dreality%2Dwill%2Dsoon%2Dbe%2Dworth%2D150%2Dbillion%2Dhere%2Dare%2Dthe%2Dmajor%2Dpla.
[25] vtime. https://play.google.com/store/apps/details?id=net.vtime.cardboard&hl=en.
[26] What vr could, should, and almost certainly will be within two years. http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf.
[27] x264library. http://www.videolan.org/developers/x264.html.
[28] O. Abari, D. Bharadia, A. Duffield, and D. Katabi. Cutting the cord in virtual reality. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 162–168. ACM, 2016.
[29] O. Abari, D. Bharadia, A. Duffield, and D. Katabi. Enabling high-quality untethered virtual reality. In *NSDI*, pages 531–544. USENIX, 2017.
[30] M. Alvarez-Mesa, C. C. Chi, B. Juurlink, V. George, and T. Schierl. Parallel video decoding in the emerging hevc standard. In *Acoustics, Speech and Signal Processing, 2012 IEEE International Conference on*, pages 1545–1548. IEEE, 2012.
[31] K. Boos, D. Chu, and E. Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 291–304. ACM, 2016.
[32] 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 *Proceedings of the 2016 ACM on Multimedia Conference*, pages 655–659. ACM, 2016.
[33] 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 Transactions on Circuits and Systems for Video Technology*, 22(12):1827–1838, 2012.
[34] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient parallelization of h. 264 decoding with macro block level scheduling. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1874–1877. IEEE, 2007.
[35] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.
[36] 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 *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.
[37] 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 *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135. ACM, 2015.
[38] V. R. Gaddam, M. Riegler, R. Eg, C. Griwodz, and P. Halvorsen. Tiling in interactive panoramic video: Approaches and evaluation. *IEEE Transactions on Multimedia*, 18(9):1819–1831, 2016.
[39] A. Garcia-Saavedra, P. Serrano, A. Banchs, and G. Bianchi. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 169–180. ACM, 2012.
[40] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: code offload by migrating execution transparently. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 93–106, 2012.
[41] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder. Foveated 3d graphics. *ACM Transactions on Graphics*, 31(6):164, 2012.
[42] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.
[43] 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 *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM, 2015.
[44] W. Li, D. Wu, R. K. Chang, and R. K. Mok. Demystifying and puncturing the inflated delay in smartphone-based wifi network measurement. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 497–504. ACM, 2016.
[45] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 7–ff. ACM, 1997.
[46] A. Mittal, A. K. Moorthy, and A. C. Bovik. No-reference image quality assessment in the spatial domain. *IEEE Transactions on Image Processing*, 21(12):4695–4708, 2012.
[47] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 113–128. ACM, 2003.
[48] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proc. of ACM Symposium on Operating System Principles*, 1997.
[49] D. Ochi, Y. Kunita, K. Fujii, A. Kojima, S. Iwaki, and J. Hirose. Hmd viewing spherical video streaming system. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 763–764. ACM, 2014.
[50] F. Qian, L. Ji, B. Han, and V. Gopalakrishnan. Optimizing 360 video delivery over cellular networks. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 1–6. ACM, 2016.
[51] B. Reinert, J. Kopf, T. Ritschel, E. Cuervo, D. Chu, and H.-P. Seidel. Proxy-guided image-based rendering for mobile devices. *Computer Graphics Forum*, 35(7):353–362, 2016.
[52] R. Shea, D. Fu, and J. Liu. Rhizome: Utilizing the public cloud to provide 3d gaming infrastructure. In *Proceedings of the 6th ACM Multimedia Systems Conference*, pages 97–100. ACM, 2015.
[53] R. Shea and J. Liu. On gpu pass-through performance for cloud gaming: Experiments and analysis. In *Proceedings of Annual Workshop on Network and Systems Support for Games*, pages 1–6. IEEE Press, 2013.
[54] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui. Cloud gaming: architecture and performance. *IEEE Network*, 27(4):16–21, 2013.
[55] S. Shi and C.-H. Hsu. A survey of interactive remote rendering systems. *ACM Computing Surveys*, 47(4):57, 2015.
[56] 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 *Proceedings of the 19th ACM international conference on Multimedia*, pages 103–112. ACM, 2011.
[57] 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 *International Symposium on Multimedia*, pages 583–586. IEEE, 2016.
[58] 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 *Proceedings of the ACM Symposium on Applied Perception*, pages 7–14. ACM, 2016.
[59] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.
[60] 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 Transactions on Circuits and Systems for Video Technology*, 25(12):1988–2001, 2015.
[61] 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 *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 91–102. ACM, 2014.