Rubiks: Practical 360-Degree Streaming for Smartphones

Jian He*, Mubashir Adnan Qureshi*, Lili Qiu*, Jin Li+, Feng Li+, Lei Han+
*The University of Texas at Austin, USA, +Network Technology Lab, Huawei, China
{jianhe,mubashir,lili}@cs.utexas.edu,{mark.lijin,frank.lifeng,phoebe.han}@huawei.com

ABSTRACT

The popularity of 360° videos has grown rapidly due to the immersive user experience. 360° videos are displayed as a panorama and the view automatically adapts with the head movement. Existing systems stream 360° videos in a similar way as regular videos, where all data of the panoramic view is transmitted. This is wasteful since a user only views a small portion of the 360° view. To save bandwidth, recent works propose the tile-based streaming, which divides the panoramic view to multiple smaller sized tiles and streams only the tiles within a user's field of view (FoV) predicted based on the recent head position. Interestingly, the tile-based streaming has only been simulated or implemented on desktops. We find that it cannot run in real-time even on the latest smartphone (*e.g.*, Samsung S7, Samsung S8 and Huawei Mate 9) due to hardware and software limitations. Moreover, it results in significant video quality degradation due to head movement prediction error, which is hard to avoid. Motivated by these observations, we develop a novel tile-based layered approach to stream 360° content on smartphones to avoid bandwidth wastage while maintaining high video quality. Through real system experiments, we show our approach can achieve up to 69% improvement in user QoE and 49% in bandwidth savings over existing approaches. To the best of our knowledge, this is the first 360° streaming framework that takes into account the practical limitations of Android based smartphones.

CCS CONCEPTS

• **Human-centered computing → Mobile computing**; *Virtual reality*; • **Information systems → Multimedia streaming**;

KEYWORDS

360^o Videos, Smartphones, Rate Adaptation, Video Codecs

ACM Reference Format:
Jian He*, Mubashir Adnan Qureshi*, Lili Qiu*, Jin Li+, Feng Li+, Lei Han+. 2018. Rubiks: Practical 360-Degree Streaming for Smartphones. In *MobiSys '18: The 16th Annual International Conference on Mobile Systems, Applications, and Services, June 10–15, 2018, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3210240.3210323

1 INTRODUCTION

Motivation: Video traffic has increased at an unprecedented rate due to the popularity of video sharing websites and social media channels like YouTube and Facebook. For example, Cisco [6] estimates video traffic will constitute 82% of all consumer Internet traffic by 2021. A significant recent advance in video technology is Virtual Reality (VR) or 360° videos. It provides panoramic views to gives an immersive experience. VR content can be viewed through dedicated headsets, such as Oculus [16] and HTC Vive [13]. Users can also experience VR by placing smartphones in headsets like Google Cardboard [8] and Samsung Gear VR [17]. This is called mobile VR. Mobile VR is expected to dominate the VR market, which is expected to grow up to 25 billion USD by 2021 [19].

To watch a 360° video, a user wears a headset that blocks outside view so the user focuses only on what is being displayed on the smartphone or VR headset. 360° videos are shot using omnidirectional cameras or multiple cameras where the collected images are stitched together. The resulting effect of either approach is a video consisting of spherical images. While watching a 360° video, a user views a defined portion of the whole image, usually it is 110° along the X-axis and 90° along the Y-axis. This user's view is termed as Field-of-View (FoV). The view automatically adapts with the user's head movement. As user moves his head along any of the X, Y, or Z axis, the video player automatically updates the FoV.

Challenges: 360° videos are streamed in a similar way as regular videos. A short duration of data, typically 1-2 seconds, is requested by the client. However, 360° videos have much higher bit rates because spherical images of 360° videos contain more pixels and higher fidelity in order to provide a good viewing experience when being watched from a close distance. Typical resolution is 4K - 8K. Streaming all pixels in 360° videos is wasteful since the user only views a small portion of the video due to the limited FoV. Moreover, streaming all pixels also create significant burden for a smartphone, which has limited storage, computation resources and power.

One recent approach [39] tailored for 360° videos is the tile-based streaming. In this approach, each 360° frame is divided into smaller sized non-overlapping rectangular regions called *tiles*. Each tile can be decoded independently. The client requests those tiles that are expected to be in FoV using head movement prediction techniques. This reduces the decoding and bandwidth usage at the receiver. However, whenever there is a prediction error, the user either sees some portion of the screen to be blank or experiences rebuffering due to missing data. This can severely degrade the user QoE. Moreover, these works explore tile-based streaming either in simulation or in desktop implementation but not on smartphones. We observed in our experiments that 8K 360° videos (which are widely available on video websites like YouTube) streamed using either regular or tile-based streaming technique cannot be decoded and displayed to user in time due to resource constraints in smartphones. This limitation primarily stems from how the video data is encoded. Encoding produces independently decodable data segments that are very rich in data, so decoding them takes long time.

MobiSys '18, June 10–15, 2018, Munich, Germany

ACM ISBN 978-1-4503-5720-3/18/06…$15.00
https://doi.org/10.1145/3210240.3210323

Our approach: We first investigate the limitations of Android hardware in decoding and displaying high resolution videos like 8K. We are interested in knowing how many concurrent threads for decoding can be instantiated. We then look into popular media codecs available on Android to understand why certain media codecs are better than others for tile-based streaming. We observe that H.264, which is the most widely used media codec nowadays is ill-suited for streaming 360° video content. So we use HEVC to implement tile-based streaming since it already has a built-in tiling scheme to encode video data.

Based on our observations, we design a novel tile-based layered encoding framework *Rubiks* for 360° videos. We exploit spatial and temporal characteristics of 360° videos for encoding. Our approach splits the 360° video spatially into *tiles* and temporally into *layers*. The client runs an optimization routine to determine the video data that needs to be fetched to optimize user QoE. Using this encoding approach, we can send the video portions that have a high probability of viewing at a higher quality and the portion that has a lower probability of viewing at a lower quality. By controlling the amount of data sent, the data can be decoded without rebuffering. *Rubiks* can save significant bandwidth while maximizing the user's QoE and decoding the video in a timely manner. To the best of our knowledge, this is the first work that develops a practical streaming framework for 360° videos to explicitly account for the resource limitations in smartphones and head movement prediction error.

We implement our framework as an Android application. We use this app and a simple server to test our implementation. Our contributions can be summarized as follows.

- Using real measurements, we identify hardware and software constraints in supporting the tile-based streaming for high resolution 360° videos on smartphones.
- We develop a novel tile-based layered coding that exploits spatial and temporal characteristics of 360° videos to reduce the decoding overhead while saving bandwidth and accommodating head movement prediction error.
- We implement our idea as an app and show it can improve user QoE by 69% and save bandwidth by 35% over existing approaches for 4K videos. It provides 36% improvement in QoE and 49% in bandwidth savings for 8K videos.

Paper outline: We first provide background on 360 videos in Sec. 2. In Sec. 3, we use our measurements to identify the limitations of the existing tile-based video coding. We describe our approach in Sec. 4, and present system design in Sec. 5. We evaluate our system in Sec. 6. We review related work in Sec. 7 and conclude in Sec. 8.

2 BACKGROUND

2.1 Existing Streaming Framework

DASH [42] is widely used for video streaming over the Internet due to its simplicity and compatibility with the existing CDN servers. In this framework, a video is divided into multiple chunks with an equal play duration (*e.g.*, a few seconds). A 360° video chunk is spatially divided into equal size portions, called *tiles*, generally 15-40 tiles [26]. Each tile is encoded independently with a few bitrates.

In tile based streaming frameworks, the client only requests for a subset of tiles according to head prediction and throughput estimation. Due to independent encoding, the client can still decode that tile subset successfully. The client constructs the 360° frame based on decoded the tiles and displays the FoV on the screen.

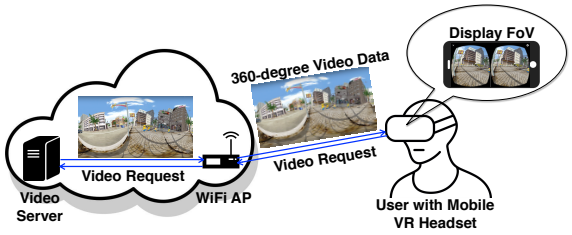


Figure 1: Architecture of Existing 360° Streaming Systems

Fig. 1 shows the architecture of existing 360° video streaming systems like YouTube and Facebook. The user sends video requests to the video server. In the request, the user has to specify which video segment and which bitrate to request. In tile based streaming frameworks, the user also needs to specify the tiles to request. The video server transmits the 360° video data to the user that is connected to the Internet via WiFi access points. The user uses a mobile VR headset to watch 360° videos. After receiving the 360° video data, the smartphone displays the video content within the FoV, which is determined by the head orientation. The performance of existing 360° streaming systems can be affected by multiple factors, including smartphone computational resources, network throughput, user head movement prediction.

2.2 H.264 and HEVC Codecs

H.264 [9] is an industry standard for video compression. Frames are encoded at the *macroblock* level. A typical macroblock size is 16×16 pixels. H.264 uses motion prediction to reduce the size of encoded data. As frames within a chunk of a few seconds are highly correlated in temporal domain, this greatly reduces the data size. HEVC [10] is an advanced extension of H.264. It divides a video frame into independent rectangular regions. Each region can be encoded independently. A region is essentially a tile used in tile based streaming framework. Each region is encoded at a 64×64 *Coding Tree Unit* (CTU) level. Due to the larger block size in HEVC, it achieves higher compression than H.264.

HEVC is more suitable for tile based streaming approaches. All tiles in HEVC are contained in a single encoded video file and can be decoded in parallel using one decoder instance. The video file is still decodable even if we remove some tiles. However, H.264 has to encode tiles into separate video files. If user requests multiple tiles, H.264 needs to decode multiple video files. The smartphone only allows a small number concurrent video decoders (*e.g.*, 4 decoders on Samsung S7 and Huawei Mate 9), which is explained in Sec. 3. When the number of video files is greater than the number of concurrent hardware decoders, some video files will have to be decoded one after another, which results in longer decoding delay. Therefore, H.264 is not scalable for tile based streaming. Even for HEVC, our measurement results in Sec. 3 show that HEVC can not decode all video tiles in real time for 8K videos. Thus, it is necessary to explore how we can avoid sending all video tiles without significant video quality degradation.

2.3 Scalable Video Coding

Scalable Video Coding (SVC) [24] is an extension of H.264. It is a layered scheme where a high quality video bit stream is composed

of lower quality subset bit streams. A subset bit stream is obtained by removing some data from higher quality bit stream such that it can be played albeit at a lower spatial and/or temporal resolution. SVC can potentially save bandwidth for tile based streaming by adapting the video quality for each tile based on the likelihood of viewing these tiles. However, currently Android does not support this extension [5].

3 MOTIVATION

In this section, we perform extensive measurements to understand the performance of existing 360° video streaming approaches on Samsung S7. We identify several significant limitations of the existing approaches and leverage these insights to develop a practical tile based streaming tailored for smartphones. First, we introduce media codecs available in Android.

3.1 Real-Time Media Codecs

There are two main options for decoding videos on Android: **ffmpeg-android** and **MediaCodec**.

ffmpeg-android: This version of ffmpeg is tailored for Android and is completely based on software. While ffmpeg supports an unlimited number of threads, they cannot decode videos in real time on smartphones because ffmpeg cannot perform hardware decoding [18]. Instead, it decodes everything in software, which is very slow. For example, decoding a 1-second video chunk with resolution 3840 × 1920 and 4 tiles takes more than 3 sec, which causes long rebuffering time.

MediaCodec: Android provides *MediaCodec* class for developers to encode/decode video data. It can access low-level multimedia infrastructure like hardware decoders, which decode the video much faster than ffmpeg-android. First, the decoder is set up based on the video format, such as H.264 or HEVC. Setting up the decoder enables access to the input and output buffers for the corresponding codec. The data that needs to be decoded is pushed into the input buffers and the decoded data is collected from the output buffers. MediaCodec is the best option for video decoding on Android as it uses dedicated hardware resources. For HEVC decoder of MediaCodec, decoding a 1-second video chunk with resolution 3840 × 1920 and 36 tiles takes around 0.5 sec.

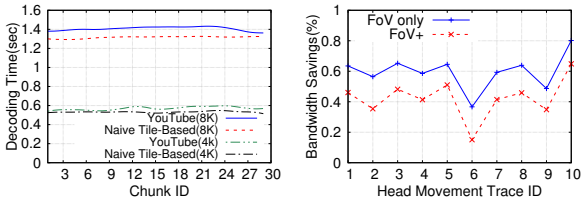
Observation: *Only hardware-accelerated media codec can support real-time decoding.*

3.2 Limitations of Existing Approaches

We focus on **MediaCodec** as ffmpeg-android is infeasible for real-time decoding. We use the following baselines in our analysis: (i) YouTube [48], which streams all data belonging to the whole 360° frames to the client, (ii) *Naive Tile-Based*, which divides 360° frames into 4 tiles and streams all tiles to the client, (iii) *FoV-only* [39], which divides the video into 36 tiles and only streams the tiles predicted to be in user's FoV. (iv) *FoV+* [23], which divides the video into 36 tiles and streams data in both FoV and the surrounding region, where the surrounding region is selected based on the estimated prediction error of FoV: if the estimated head movement prediction error is ϵ along X axis, it extends the FoV width at both sides of the X axis by ϵ; similarly for the Y axis. We quantify the performance using three metrics: (i) Decoding Time, (ii) Bandwidth Savings, and (iii) Video Quality.

In our experiments, we use HEVC as the decoder. Recent measurement [48] finds that existing 360° streaming systems (*e.g.*YouTube and Oculus) stream the entire 360° frames. H.264 can only decode one tile from an input video file, while HEVC can include multiple video tiles in the input file and decode them in parallel. Due to limited number of concurrent hardware decoders on smartphones, H.264 has to decode some tiles serially when decoding multiple video tiles. HEVC has faster decoding speed in tile based streaming frameworks than H.264 since it can decode all requested video tiles in parallel. Our baseline YouTube follows the existing 360° streaming systems to stream the entire 360° frames and uses a faster decoder available on smartphones.

3.2.1 Decoding Time. The feasibility of current approaches in streaming high quality 360° videos depends on whether they can decode the data and display it in time. To answer this question, we decode 4K and 8K videos for both YouTube and *Naive Tile-Based* approaches on smartphone. The input videos have frame rate of 30fps and 30 chunks each, where each video chunk contains 1 second video. We run our experiments 5 times for the same video.



(a) Decoding Time. (b) Bandwidth Savings

Figure 2: Decoding time and bandwidth savings.

The YouTube approach directly encodes the entire 360° chunk into a single tile. The *Naive Tile-Based* approach divides each 360° frame into 4 video tiles, and encodes them into independent video files. We run 4 video decoders simultaneously to decode video tiles in parallel. We only show the results of the *Naive Tile-Based* approach for 4 tiles, since this gives the best performance and further increasing the number of tiles can only be slower since only up to 4 decoding threads can run in parallel.

We first measure the decoding time of 4K 360° videos. The resolution of the test video is 3840 × 1920. Fig. 2(a) shows that both YouTube and *Naive Tile-Based* approaches can decode a 4K video chunk in 0.55 seconds on average. So 4K videos can be decoded and displayed in time.

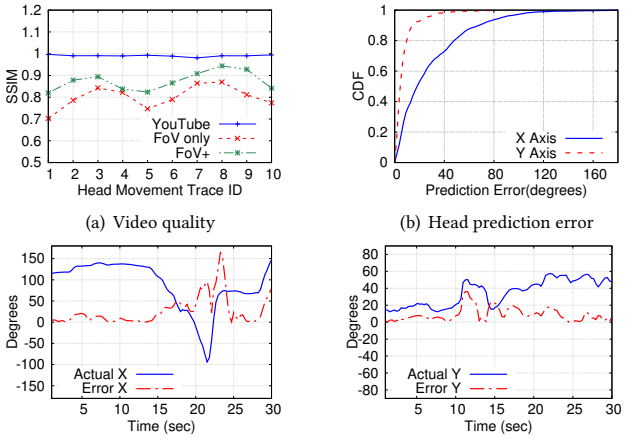
Next, we measure the decoding time of 8K 360° videos. The resolution of the test video is 7680 × 3840. Fig. 2(a) shows the average decoding time of each chunk. YouTube needs 1.4 sec on average to decode one chunk, which results in rebuffering since decoding speed cannot catch up with the playback speed. The *Naive Tile-Based* approach needs 1.3 sec on average to decode one chunk. It speeds up decoding by utilizing parallel threads. However, it is still insufficient to support real-time decoding for 8K videos. Using parallel threads cannot reduce the amount of data to read from the video decoder output buffers, which limits the performance gain from parallel threads.

Observation: *Traditional decoding and existing tile-based decoding cannot support 8K or higher video resolution.*

3.2.2 Bandwidth Savings. YouTube wastes lots of bandwidth because it streams the entire frames while a user views only a small portion. *FoV-only* and *FoV+* can save bandwidth. Fig. 2(b) shows the bandwidth savings of *FoV-only* and *FoV+* compared with YouTube. We test the bandwidth savings of the same video and 10 different head movement traces. For *FoV-only*, we use the oracle head movement, calculated from gyroscope readings, to estimate the maximum possible bandwidth savings. The video bitrate is set to a constant value for all chunks and approaches. The tiles with high motion have larger size than those with smaller motion. If a user views the tiles with larger motion, bandwidth saving is less. Since users have different viewing behaviors, we observe different bandwidth savings across head movement traces. *FoV-only* approach can save bandwidth by up to 80% but may incur significantly video quality degradation due to prediction error. *FoV+* approach saves 18% less bandwidth than *FoV-only* approach, but incurs smaller degradation in video quality.

Observation: *Due to the limited FoV, significant amount of bandwidth can be saved.*

3.2.3 Video Quality. To save bandwidth, the tiles outside the predicted FoV are not streamed in the *FoV-only* approach. We use linear regression to predict head movement. As explained in Sec. 6, our predictor uses the head movement in the past 1-second window to predict head movement for the future 2-second window. When the head prediction has large error, the user sees blank areas. This results in a very poor viewing experience. In the *FoV+* approach, additional tiles are streamed to account for head movement prediction error, but this error estimation itself can be inaccurate.



(a) Video quality (b) Head prediction error

(c) Head trace (X Axis) (d) Head trace (Y Axis)

Figure 3: Video quality of existing approaches

We quantify the video quality using SSIM [45], defined as the structural difference between streamed and original user's FoV. Fig. 3(a) shows the SSIM for 10 different head movement traces and an example 4K video. The video bitrate is set to the highest value in our experiments. YouTube can always achieve SSIM close to 1, while the average SSIM of *FoV-only* and *FoV+* are only 0.77 and 0.87, respectively. The average prediction error along the X axis and Y axis for the test head movement traces is 30 and 9 degrees, respectively. On average, our head movement traces have prediction error larger than 50° (or 10°) along the X axis (or Y axis) for around 20% of time. In the *FoV-only* approach, we observe that 20% chunks have 23% blank areas on average along the X axis and 18% along the Y axis. By including extra tiles, the *FoV+* approach can reduce blank areas along the X axis to 14% and along the Y axis to 11% on average. Thus, sending extra data can still not avoid blank areas since the head movement prediction error is unpredictable, as well. Fig. 3(c) and Fig. 3(d) show one example head movement trace and its corresponding prediction error. We can observe that when head moves quickly, the prediction error goes up significantly. For example, the prediction error for the X axis increases to 100 degrees at time 22 sec and that for Y axis increases to 40 degrees at time 12 sec. We find that fast head movement mainly happens when the user randomly explores different scenes in the video or follows an interesting fast-moving object.

Observation: *Streaming a few extra tiles is not robust enough to head movement prediction error.*

3.3 Insights From Existing Approaches

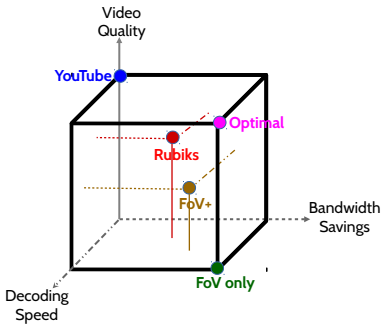


Figure 4: Design Space of Streaming Algorithms

Figure 4 summarizes the existing algorithms. *YouTube* achieves the highest video quality but at the expense of bandwidth and decoding speed. *FoV-only* and *FoV+* save bandwidth and increase decoding speed, but suffer from degraded video quality. A desirable algorithm should simultaneously optimize all three metrics: bandwidth saving, decoding speed, and video quality.

An important design decision is how to encode 360° videos to optimize these three metrics. Such a scheme should (i) adapt the data to stream based on the FoV to save bandwidth, (ii) support fast decoding using a limited number of threads, and (iii) tolerate significant head movement prediction error. (i) suggests tile-like scheme is desirable. (ii) suggests we do not have the luxury to allocate a tile to each decoding thread, but should have a different task-to-thread assignment. (iii) suggests we should still stream entire video frames albeit at a lower resolution in case of unpredictable head movement.

4 OUR APPROACH

We propose a novel encoding framework for 360° videos to simultaneously optimize bandwidth saving, decoding time, and video quality. Each video chunk is divided spatially into *tiles* and temporally into *layers*. The two dimensional splitting allows us to achieve the following two major benefits.

- We can stream different video portions at different bitrates and with different numbers of layers. The video portions with a high probability of viewing are streamed at a higher quality. The ones with a lower chance of viewing are sent at

a lower quality instead of not sending them at all. This allows us to save network bandwidth while improving robustness against head movement prediction error.
- By managing the amount of data sent for tiles, we can control the decoding time. In 8K videos, we can not decode all tiles in time due to hardware constraints, so we can selectively send tiles with less viewing chance to improve efficiency.

The performance of 360° video streaming is determined by video coding and video rate adaptation. Below we examine them in turn.

4.1 Video Encoding

We propose a tile-based layered video encoding scheme. A 360° video chunk is spatially divided into tiles, which are further temporally split into layers. We utilize redundant I-Frames to remove encoding overhead due to layering.

Spatial Splitting: As shown in Fig. 5, in spatial domain, each 360° frame can be divided into multiple equal-size regions, called *tiles*. These tiles are encoded independently so that each tile can be decoded individually. Each tile is encoded at several bitrates for a client to control the quality of the tiles.

Temporal Splitting: Each tile consists of N frames distributed across M *layers*. The base layer includes the first of every M frames, the second layer includes the second of every M frames, and so on. For example, for a video chunk consisting of 16 frames, frames 1, 5, 9, 13 form the base layer; frames 2, 6, 10, 14 form the second layer; frames 3, 7, 11, 15 form the third layer; and frames 4, 8, 12, 16 form the fourth layer. Fig. 5 shows how temporal splitting is applied to each tile. We consider the highlighted tile. Each tile can be decomposed into M layers by distributing N frames as described above. n_{tl} denotes t-th tile at the l-th layer. This is the granularity at which we encode video data.

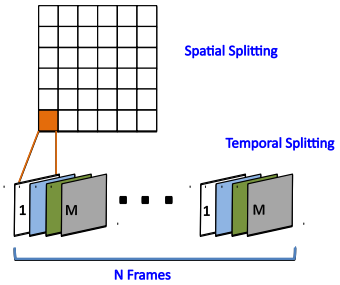


Figure 5: Spatial and temporal splitting of 360° chunk

Reducing Encoding Overhead: In an encoded video file, the first frame is encoded as an I-frame, which is decoded independently and is large in size. The subsequent frames are encoded as B-frames or P-frames, which reference neighboring frames to significantly reduce the frame size. We generate M independently decodable video files corresponding to M layers of each chunk, and each layer has a separate I-frame. In comparison, the YouTube approach generates a single I-frame since it encodes all data in a single file.

To eliminate this coding overhead, we remove the I-frames from the enhancement layers as follow. We insert the first frame from the base layer to the beginning of each enhancement layer. After encoding, the I-frame from each enhancement layer can be removed since it is the same as the first frame in the base layer. When decoding video, we just need to copy the I-frame from the base layer to the beginning of each encoded enhancement layer, thereby reducing video size by removing the redundant I-frames.

4.2 360° Video Rate Adaptation

Despite significant work on rate adaptation, the 360° video rate adaptation is a new and under-explored topic. Unlike the traditional rate adaptation, where the user views the entire frame, a user only views a small portion of 360° videos. Therefore, the high-level problem is to select which portion to stream and at what rate to maximize the user QoE. This is challenging because of unpredictable head movement. We use an Model Predictive Control (MPC) based framework [47] which is efficient to optimize the user QoE even if network throughput fluctuation is unpredictable. Our optimization takes the following inputs: predicted FoV center, estimated FoV center prediction error, predicted throughput and buffer occupancy. It outputs the number of tiles in each layer and the bitrate of each tile. We first introduce our MPC framework, and then describe how to compute each term in the optimization objective.

4.2.1 MPC-based Optimization Framework. To handle random throughput fluctuation, our optimization framework optimizes the QoE of multiple chunks in a future time window. Given the predicted network throughput during the next w video chunks, it optimizes the QoE of these w video chunks. The QoE is a function of the bitrate, the number of tiles to download for each layer, and the FoV. This function can be formulated as follow:

$$\max_{(\boldsymbol{r_i},\boldsymbol{e_i}),i\in[t,t+w-1]} \sum_{i=t}^{i=t+w-1} QoE_i(\boldsymbol{r_i},\boldsymbol{e_i},\boldsymbol{c_i}) \tag{1}$$

where QoE_i denotes the QoE of chunk i ,w denotes the optimization window size, $\boldsymbol{r_i}$ denotes the bitrate of tiles to download for chunk i, $\boldsymbol{e_i}$ is a tuple whose element e_i^j denotes the number of tiles to download for the j-th layer in the i-th chunk. Since a user's FoV varies across frames in a chunk, we explicitly take that into account by computing the QoE based on $c_i^k = (x_i^k, y_i^k)$, which denotes the X and Y coordinates of the FoV center of frame k in chunk i. We search $(\boldsymbol{r_i},\boldsymbol{e_i})$ that maximizes the objective within the optimization window, and then request the data for chunk t according to the optimal solution. In the next interval, we move the optimization window forward to $[t+1, t+w]$ to optimize the next chunk $t+1$.

4.2.2 User QoE. Next, we define the user QoE metric. It is widely recognized that the user perceived QoE for a video chunk is determined by the following factors: video quality, quality changes, and rebuffering time [36, 43, 47].

Video quality: Each video chunk has K frames. The quality of frame k in chunk i is a function of bitrate $\boldsymbol{r_i}$, number of tiles to download $\boldsymbol{e_i}$, and the FoV center c_i^k, We let $h(\boldsymbol{r_i},\boldsymbol{e_i},c_i^k)$ denote the video quality. By averaging quality across all frames in the current chunk, we get the quality of chunk i as follow:

$$f_1(\boldsymbol{r_i},\boldsymbol{e_i},\boldsymbol{c_i}) = \frac{1}{K}\sum_{k=1}^{K} h(\boldsymbol{r_i},\boldsymbol{e_i},c_i^k) \tag{2}$$

Note that different frames may have different FoV, so their quality is determined by their corresponding FoV center c_i^k.

Quality changes: The video quality changes between two consecutive chunks is defined as

$$f_2(\boldsymbol{r_i}, r_{i-1}, \boldsymbol{e_i}, \boldsymbol{e_{i-1}}, \boldsymbol{c_i}, \boldsymbol{c_{i-1}}) = |f_1(\boldsymbol{r_i}, \boldsymbol{e_i}, \boldsymbol{c_i}) - f_1(r_{i-1}, \boldsymbol{e_{i-1}}, \boldsymbol{c_{i-1}})| \quad (3)$$

where $(\boldsymbol{r_i}, \boldsymbol{e_i})$ and $(\boldsymbol{r_{i-1}}, \boldsymbol{e_{i-1}})$ represent the bitrates and numbers of tiles to download for chunk i and chunk $i-1$, respectively.

Rebuffering: To compute the rebuffering time, we observe that the chunk size depends on the bitrate and the set of tiles downloaded. Let $v_i(\boldsymbol{r_i}, \boldsymbol{e_i})$ denote the chunk size. We start requesting the next chunk after the previous chunk has been completely downloaded. The buffer occupancy has a unit of second. Let B_i denotes the buffer occupancy at the time of requesting chunk i. Each chunk contains L-second video data. Let W_i denote the predicted network throughput of downloading chunk i. Then, the buffer occupancy when requesting chunk $i+1$ can be calculated as:

$$B_{i+1} = \max(B_i - \frac{v_i(\boldsymbol{r_i}, \boldsymbol{e_i})}{W_i}, 0) + L \quad (4)$$

The first term indicates that we play $\frac{v_i(\boldsymbol{r_i}, \boldsymbol{e_i})}{W_i}$ sec video data while downloading chunk i. Afterwards, L-sec video data will be added to the buffer. The rebuffering time of chunk i is

$$f_3(\boldsymbol{r_i}, \boldsymbol{e_i}) = \max(\frac{v_i(\boldsymbol{r_i}, \boldsymbol{e_i})}{W_i} - B_i, 0) + \max(\tau_i - L, 0) \quad (5)$$

where τ_i denotes the decoding time of chunk i and is derived from our measurement. The first term denotes the rebuffering time incurred due to slow downloading and the second term denotes the rebuffering time incurred due to slow decoding. The expression considers the fact that the chunk i's decoding starts after finishing playing out everything ahead of the chunk. Note that we can start playing a chunk even if it is only partially decoded.

Putting them together, we compute the QoE of chunk i as follow:

$$QoE_i(\boldsymbol{r_i}, \boldsymbol{e_i}, \boldsymbol{c_i}) = \alpha f_1(\boldsymbol{r_i}, \boldsymbol{e_i}, \boldsymbol{c_i}) - \beta f_2(\boldsymbol{r_i}, r_{i-1}, \boldsymbol{e_i}, \boldsymbol{e_{i-1}}, \boldsymbol{c_i}, \boldsymbol{c_{i-1}}) - \gamma f_3(\boldsymbol{r_i}, \boldsymbol{e_i}) \quad (6)$$

where α, β and γ are weights of video quality, quality changes and rebuffering, respectively. The latter two terms are negative since we want to minimize them.

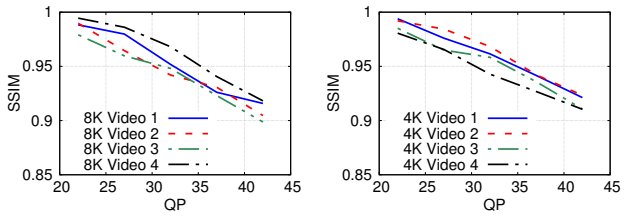
4.2.3 Estimate Video Quality. The first and second terms in the user QoE are determined by the video quality. To support efficient optimization, we need to quickly compute the video quality for a given streaming strategy. That is, for a given FoV, we need to determine the video quality of streaming a selected set of tiles in all layers and bitrates.

Video Quality Metric Approximation: In our optimization problem, we need to estimate the video quality of the predicted FoV. Exactly computing video quality metrics for all paths in the optimization problem is too expensive. Moreover, since user head movement is not known in advance, computing video quality offline requires computing for all possible FoV, which is also too expensive. We develop an efficient scheme to approximate video quality metric.

Before we introduce our algorithm, first we describe how video is constructed from various layers. For each tile, we extract the data from all layers associated with the tile. As described in Section 4.1, we divide videos temporally into layers and a layer j corresponds to the j-th frame in every 4 frames. Therefore, given a tile that has k layers, we put the downloaded tiles for these k layers at the corresponding positions and add the missing frame by duplicating the last received layer. For example, if a tile only has a base layer, we duplicate the base layer 3 times for every group of 4 frames. If a tile has the first 3 layers, we use the data from the layers 1, 2 and 3 to form the first three frames and duplicate the third frame to form the frame 4 every group. According to the above video construction, we derive the following metric based on the observation that there exists a strong correlation between video quality and bitrate. This indicates an opportunity of using video bitrate for optimization. Quantization parameter (QP) [11] is used by HEVC to control video bitrate. A larger quantization indicates a lower bitrate. The quality of frame k in chunk i, denoted as $h(\boldsymbol{r_i}, \boldsymbol{e_i}, c_i^k)$, is defined as follow:

$$h(\boldsymbol{r_i}, \boldsymbol{e_i}, c_i^k) = \frac{1}{|FoV(c_i^k, \boldsymbol{e_i})|} \sum_{l \in FoV(c_i^k, \boldsymbol{e_i})} q(r_i^l, \boldsymbol{e_i}) \quad (7)$$

where $FoV(c_i^k, \boldsymbol{e_i})$ denotes the set of tiles within the FoV centered at c_i^k and $q(r_i^l, \boldsymbol{e_i})$ represents the quality of tile l in the FoV. $h(\boldsymbol{r_i}, \boldsymbol{e_i}, c_i^k)$ averages the quality over all tiles in the FoV and the quality of each tile is determined by the number of layers streamed for the tile and the data rate it is streamed at.



(a) 8K Videos (b) 4K Videos

Figure 6: Correlation between video quality and bitrate.

For the tiles in the predicted FoV, they have the highest probabilities of being viewed. Therefore, we stream all layers of the tiles in the predicted FoV, and evaluate $q(r_i^l, \boldsymbol{e_i})$ assuming 4 layers.

We study the correlation between r_i^l and SSIM. We set r_i^l to different QP values and the number of layers is set to 4. The input video is divided into 36 tiles. We set the FoV center to the center of video tiles to calculate the video quality under 36 different FoVs. FoV quality is the average SSIM of the same FoV across all video chunks. Fig. 6 shows the correlation between average FoV quality and QP for both 4K and 8K videos. We observe that average FoV quality decreases linearly with the normalized video QP. The average correlation coefficient among all test videos is 0.98. We approximate FoV quality using $-0.004 \times \text{QP} + b$. From Fig. 6, we can see that the value of b varies across different videos. However, b remains constant for all chunks of the same video. We set b to 0 since removing a constant in object function does not affect the optimization solution.

4.2.4 Decoding Time. The third term in the user QoE is the rebuffering time, which is affected by both the downloading time and decoding time. Existing rate adaptation scheme ignores the decoding time and only uses the downloading time to determine the rebuffering time. This is acceptable for regular videos with much fewer pixels and fast desktops. But decoding time for 360° videos

on smartphones can be significant and sometimes exceed the downloading time. Therefore it is important to estimate the decoding time for different streaming strategies to support optimization.

Decoding time depends on the number of tiles in each layer. Moreover, there is also variation in decoding time even when decoding the same tile configuration due to other competing apps on a smartphone. We model this decoding time and take into account the variation.

To model the decoding time, we measure the decoding time for 8 videos by varying the bitrate and the number of tiles in each layer. We decode each configuration 3 times and record the results. As we have 4 threads, so each measurement has 4 decoding time values and the overall decoding time is dictated by the thread that takes the longest time. Each thread decodes one layer. An underlying assumption here is that all decoding threads start at the same time. We observe that there is not a significant variation in the start time of these threads, so this assumption works well in practice.

We implement a simple table lookup for decoding time based on measurement, where the table is indexed by (# tiles layer-2, # tiles layer-3, # tiles layer-4). We do not consider the number of tiles in the base layer as it always contains all tiles. Moreover, we also do not consider the video bitrate because we observe that it does not impact the decoding time significantly. This is because the bit rate only affects the quantization level, while the video decoding complexity mainly depends on the resolution of the input video. The decoding time entries in the table are populated by averaging the maximum decoding time of each instance for all measurement sets. We use a simple table lookup for decoding time because the variation in decoding time for decoding the same configuration is not large (*e.g.*, within 7% and 6% on average for Samsung S7 and Huawei Mate9, respectively).

4.2.5 Improving Efficiency. The large search space poses a significant challenge for real-time optimization. To support efficient optimization, we identify the following important constraints that can be used to prune the search space.

Constraints on the numbers of tiles: $e_i^j \geq e_i^{j'}$ for any layer $j < j'$. This is intuitive as the lower layer tiles should cover no smaller area to tolerate prediction errors.

Constraints on the bitrates: $r_i^l \geq r_i^{l'}$ for any tiles $l \in FoV(c_i^k, \boldsymbol{e_i})$ and $l' \notin FoV(c_i^k, \boldsymbol{e_i})$, where r_i^l denotes the bitrate of tile l for chunk i. This means that tiles which outside the predicted FoV should have no higher bitrate since they are less likely to be viewed.

Temporal constraints: When the throughput is stable over the optimization window (*i.e.*, no significant increasing or decreasing trend), all future chunks have the same streaming strategy (*i.e.*, $(\boldsymbol{r_i}, \boldsymbol{e_i}) = (\boldsymbol{r_{i'}}, \boldsymbol{e_{i'}})$, where i and i' are any of the future w chunks.

5 SYSTEM DESIGN

5.1 System Architecture

Fig. 7 shows two major components of our system. (i) the *Client side* estimates the predicted FoV and network throughput, runs the optimization, and generates requests accordingly, and (ii) the *Server side* handles the video encoding (*i.e.*, including spatially partitioning into tiles and temporally splitting into layers) and stream data according to the client requests.

The *Client side* runs the optimization, which takes the head movement prediction of the user, prediction error, playback buffer and network throughput as the input and outputs the data that needs to be requested next. The objective is to maximize the user QoE while taking into account the network throughput and decoding time of incoming data.

The red arrows in Fig. 7 indicate the complete process from video chunk request generation to chunk playback.

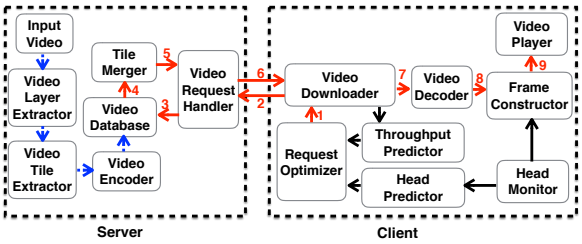


Figure 7: System Architecture

5.2 Server Side

Fig. 7 shows different modules of our server. We use the standard equirectangular projection to store raw 360° frames, which is used by YouTube [20]. There are other ways to store raw 360° frames like cubemap [7] proposed by Facebook. The cubemap projection tries to reduce the size of raw 360° video without degrading video quality. *Rubiks* focuses on how to transmit tiles in the projected 360° frames. Projection methods can not speed up video decoding since it does not reduce the video resolution which determines decoding speed.

Video Layer Extractor and *Video Tile Extractor* divide the video data spatially and temporally as described in Sec. 4. We use 36 tiles and 4 layers in our implementation. We use an open-source HEVC encoder kvazaar [14] for encoding. We let kvazaar restrict motion compensation within video tiles such that each tile can be decoded independently. Encoded video data is stored in a video database.

When the video request handler receives a request, it queries the video database to find the requested tiles. The video database sends the requested tiles to the tile merger to generate a merged chunk for each layer. We spatially merge the independently decodable tiles from the same layer into a video chunk file. The client can decode the portion of 360° view covered by the tiles contained in the merged video chunk. Since the client needs the size of video tiles to optimize video requests, the video request handler sends tile size as the meta-data before sending the encoded video data.

5.3 Client Side

As shown in Fig. 7, the client first predicts head movement, and uses this information along with the network throughput to perform 360° video rate adaptation. Then it requests the corresponding video from the server, decodes each layer, merges all layers, and finally plays the resulting video. Next we describe each module.

Tracking and predicting head position: We need head movement prediction to determine which part of the video the user is going to view in the next few seconds. When watching a 360° video, the user head position can be tracked using gyroscope, which is widely available on smartphones. Note that, the head position estimated from gyroscope readings is considered as the ground-truth

head position in our system. We can then use this head position to get the center of FoV. X-axis can go from -180° to 180°, Y-axis and Z-axis can go from -90° to 90°. Head movement exhibits strong auto-correlation on small time scales [23]. So we use past head motion to predict future head motion.

We collect head movement traces from 20 users for 10 videos, each lasting around 30 seconds. We randomly select half of the head movement traces as the training data and use the other half for testing. Sampling interval of gyroscope measurements is set to 0.1 sec. We use least square to solve $Y = AX$, where X is the past head movement and Y is the future movement. We learn A using the training data X and Y. We apply the learned A and past head movement X to predict the future movement Y. Moreover, we also estimate the error of our head movement prediction using the least square and use the error to determine additional tiles to request for robustness. We train a separate model for each of the three axes. In our evaluation, we use the past 1-second head movement to predict the future 2-second movement. In our system, the time to predict the head position and error is only 2ms.

Network throughput predictor: The client continuously monitors the network throughput when downloading video data. It records the network throughput in the previous 10-sec window, and uses the harmonic-mean predictor [32] to predict the network throughput in the next optimization window.

Video request optimizer : Given the predicted head position and throughput, the optimization searches for the optimal decision. The decision specifies the set of tiles to download from each layer and their corresponding bitrates. We implement the optimization routine using Android JNI since it is much faster than general Java. The optimization window size w is set to 3. Due to the small search space, it can finish the optimization within 12ms on average.

Video downloader : It maintains HTTP connections with the server to download video. The optimization results are used to construct HTTP requests.

Video decoder : We exploit hardware-accelerated Android media codec to decode video data. Four parallel video decoders are initialized before we start playing video. Note that four threads are the maximum number of concurrent decoders we can run due to limited hardware resource. Each video layer has one merged video chunk file. So each decoder decodes all the requested tiles of one layer. When decoding a video chunk file, the corresponding video decoder will run decoding as a background thread, which is implemented using Android AsyncTask. Decoding has to run in the background to avoid blocking video playback in the UI thread.

Frame constructor : 360° frames are reconstructed based on downloaded tiles from each layer.

Video Player : 360° video frames constructed from the frame constructor are rendered through Android OPENGL ES, which uses GPU to render video frames. The head monitor tells the video player which portion of the 360° should be displayed.

6 EVALUATION

We implement *Rubiks* as an Android app and run our experiments on Samsung Galaxy S7 and Huawei Mate9. We show that *Rubiks* can not only support 8K videos on smartphones, which is infeasible for the existing tile based streaming approaches, but also enhances user experience and saves bandwidth for both 4K and 8K videos.

6.1 Evaluation Methodology

In this section, we first introduce the experiment setup. Then, we explain the experiment configuration.

6.1.1 Experiment Setup. We run our video server on a laptop equipped with a wireless NIC Intel AC-8260. The client runs on Samsung Galaxy S7 and Huawei Mate 9. We configure the laptop as an access point such that the smartphone can have wireless connections to the video server through WiFi. The smartphone remains close (e.g. <1m) to the laptop such that it always has stable wireless connections to the laptop. We use *tc* [15] to control the throughput between the smartphone and laptop. For trace-driven experiments, video player renders the video according to the user FoV in the head movement traces. For user study, both head movement prediction and FoV rendering are based on actual gyroscope readings. We quantify the performance of *Rubiks* using the user QoE as we vary the videos, head movement traces and network throughput traces. We also examine individual terms in the QoE metric, including video quality, video quality changes, rebuffering and bandwidth savings.

6.1.2 Experiment Settings.

Video traces : We test the performance of our system for both 4K and 8K videos. We download 16 videos from YouTube: eight 4K videos and eight 8K videos. The resolution of 4K video is 3840×1920, while the 8K resolution is 7680 × 3840. Each video is divided into 30 video chunks, where each chuck has 32 frames. For 4K videos, 4 of them have fast motion (*e.g.*, videos of football games [1], roller coaster [2], etc.), while the other 4 have slow motion (*e.g.*, videos of underwater scene [4], sailing [3], etc.). 8K videos have similar motion characteristics.

A video chunk is further divided into 4 layers. Each layer divides a 360° frame into 6×6 uniform tiles. For 4K videos, the resolution of a video tile is 640 × 320. A video tile from 8K videos has resolution 1280 × 640. We use Quantization Parameter(QP) to specify the encoding bitrate of videos. The QP value has options: 22, 27, 32, 37 and 42, recommended from the recent work [26].

Throughput traces : We select 10 throughput traces from the dataset HSDPA [12] with varying throughput patterns. To ensure sufficient throughput to support 4K and 8K videos, we scale up the traces according to the bitrate of test videos. For 4K videos, we scale the traces to have average throughput in the range 0.15MBps-2.1MBps. For 8K, we scale to the range 3MBps-27MBps.

Head movement traces : We collect real head movement traces when users watch our test 360° videos via a headset Samsung Gear VR. For each video, we collect 20 head movement traces. We randomly select half of the traces to learn the weights of the linear predictor. The trained linear predictor is used by the head predictor module to estimate head movement of users when watching a 360° video. We use the other half of traces as practical user head movement behaviors to evaluate the performance of *Rubiks*.

Video quality metric : In our experiment results, we show actual video quality via SSIM [45] since it has high correlation with actual user QoE. It is defined as structural difference between the streamed and actual FoV. A higher SSIM indicates higher user QoE.

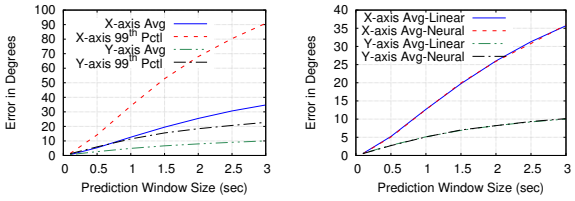
QoE weights : We set the weights in QoE definition as follows, $\alpha = 1$, $\beta = 0.5$ and $\gamma = 1$, which is a commonly used setting in the existing works [36, 43, 47].

6.2 Micro Benchmarks

In this section, we quantify the head movement prediction error and decoding time modeling error.

6.2.1 Head Movement Prediction Error. We want to understand how head movement prediction error changes when we vary (i) the future window in which to predict head movement, which is called as *prediction window*, (ii) the algorithm used for prediction, and (iii) the amount of historical information used for prediction. We compare two algorithms: Neural Networks and Linear Regression. In our analysis, we use 100 head movement traces, collected from 10 users by showing them 10 videos. We train the model using 30% data and test on 70% data. Only X and Y axis movement is considered because most of the head movement is along these dimensions.

Figure 8(a) shows how the prediction error changes along the X and Y axes when we vary the prediction window size. As expected, the prediction error increases with the prediction window size. For a 2-sec prediction window that we use, the average prediction errors along the X and Y axes are 25° and 8°, respectively, and the 99^{th} percentile errors are 60° and 20°, respectively. Traditional tile based streaming suffers from poor viewer quality experience when the prediction error is very large, because the user sees either black screen or waits for fetching of absent data, which incurs rebuffering. In our traces, the maximum error along X-axis is 170° and along Y-axis is 69°. However, using *Rubiks*, user can still see the content outside the predicted FoV, so it is more robust to large head movement prediction errors.



(a) Vary Prediction Window (b) Regression vs Neural Net.

Figure 8: Head Movement Prediction Analysis

We compare the linear regression with a non-linear modeling tool: Neural Networks. We train a neural network with 3 hidden layers and each layer contains 30 neurons. As shown in Figure 8(b), there is no performance difference between Neural Networks and Linear Regression. So we opt for linear regression for its simplicity. We also vary the historical window size while fixing the prediction window to 1 sec. We observe that past 0.5 seconds are enough to predict the future head movement because head movements in distant past are not highly correlated with future movement. The distant head position variables are given low weights so we use past 1 sec historical information.

6.2.2 Decoding Time Modeling. We evaluate how decoding time is affected when we use different tile configurations. This helps estimate the maximum number of tiles that can be decoded in time. Moreover, we also quantify the accuracy of our model. We measure the decoding time of eight 8K videos. It is not necessary to model

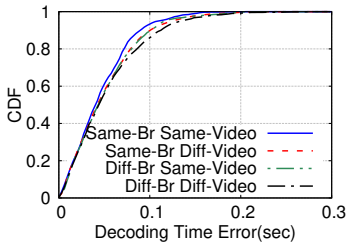


Figure 9: Modeling Error.

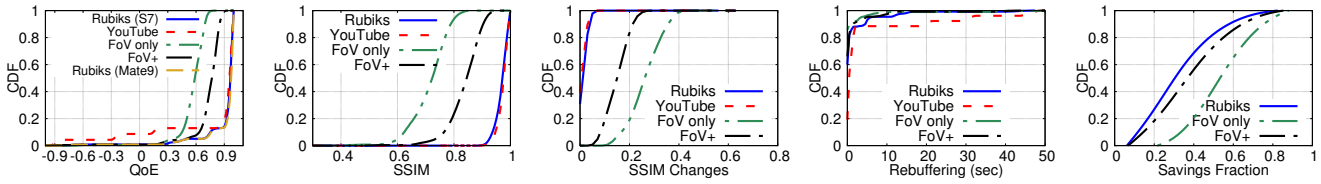
decoding time for 4K videos, since all of our testing algorithms can decode them in real time. For tile configurations, we decode all tiles for the base layer. The number of tiles for each enhancement layer can take any value from the list {9, 12, 16, 20, 25, 30}. The test videos have 5 different bitrates. For a specific bitrate and tile configuration, we run the decoding time experiments 3 tiles. We avoid measuring the decoding time of tile configurations whose lower layers have fewer tiles than higher layers since that is not practical.

Table 1 shows a few example entries in the decoding time lookup table trained on Samsung S7 and Huawei Mate9. Note that, the tile configuration tuple indicates the number of tiles from layer 2, 3 and 4, respectively. The number of tiles from layer 1 is fixed at 36. There are 56 entries in the lookup table. We populate the lookup table by averaging the decoding time across videos with the same bitrate and tile configuration. We observe that the maximum number of tiles which can be decoded in real time is (25, 20, 20). To evaluate the impacts of new hardware on decoding time, we also include the decoding time of Samsung S8 which is equipped more recent hardware. We find that S8 has very similar decoding time as Mate9. In Sec. 6.4, we will discuss that *Rubiks* has significant improvement in video quality and bandwidth savings, compared with existing state-of-the-art algorithms, in addition to speeding up decoding.

Tile Conf.	16,12,9	20,12,9	25,16,9	25,16,12	25,20,20
S7	0.71s	0.79s	0.91s	0.95s	1.05s
Mate9	0.68s	0.75s	0.87s	0.90s	1.03s
S8	0.66s	0.74s	0.85s	0.89s	1.02s

Table 1: Decoding Time Lookup Table

Fig. 9 shows the CDF of decoding time modeling error for the configuration (25, 16, 12) in following cases: (1) applying the model to the same video at the same bitrate, (2) applying the model to the same video with a different bitrate, (3) applying the model to a different video at the same bitrate, (4) applying the model to a different video at a different bitrate. We build the lookup table from one video to test the modeling accuracy across different videos. We can see that 90^{th} percentile error for all cases is less than 0.1sec. To handle this modeling error, we inflate the decoding time by 0.1 sec when the optimization routine estimates rebuffering time based on decoding time for a given strategy. Since cross-video and cross-bitrate cases do not increase modeling error, we can populate the lookup table with similar modeling error as shown in Fig. 9 by measuring decoding time for a single bitrate and one 30-sec video. Thus, it is easy to generate a lookup table for a new phone within half an hour. We can generate decoding tables for different phones and store them in app database. A user downloads all these tables alongside the app, the app can then choose appropriate table based on the user's smartphone model.



(a) QoE (b) Video Quality (c) Quality Changes (d) Rebuffering (e) Bandwidth Savings

Figure 10: Performance of 4K videos(8 videos, 10 throughput traces, 80 head movement traces)

6.3 System Results

In this section, we evaluate our system. We compare it against the following three baseline schemes: (i) *YouTube* [48]: Streaming the 360° as a single-tile video. (ii) *FoV-only* [39]: The video is divided into 6×6 uniform tiles. We only stream tiles in the predicted FoV. (iii) *FoV+* [23]: We enlarge the predicted FoV to include the surrounding parts according to the prediction error and stream the content in the enlarged FoV.

6.3.1 Rubiks *for 4K videos.*

User QoE: Fig. 10(a) shows on average *Rubiks* out-performs YouTube, *FoV-only* and *FoV+* by 14%, 69%, and 26%, respectively. All schemes can decode 4K videos in real-time. *Rubiks* improves QoE over YouTube because it reduces the amount of data to send and leads to less rebuffering. *Rubiks* improves over *FoV-only* and *FoV+* due to its robustness to head movement prediction errors.

We test the performance of *Rubiks* on both Samsung S7 and Huawei Mate 9. Because both phones can support real-time decoding for 4K videos, the difference between QoE achieved by two phones is within 1%, as shown in Fig. 10(a).

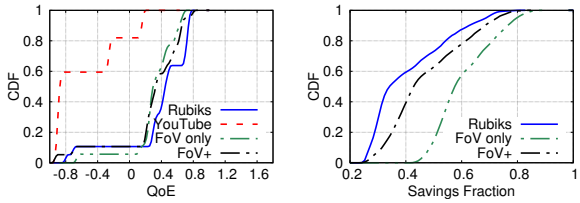
Rebuffering: *Rubiks* incurs less rebuffering time since it sends much less data than YouTube. Fig. 10(d) shows that the average rebuffering time of *Rubiks* and YouTube is 1.2 sec and 4.1 sec, respectively. The reduction in rebuffering time accounts for most of the QoE improvement in *Rubiks*.

Video Quality: Fig. 10(b) shows that the average video quality of *Rubiks*, YouTube, *FoV-only*, and *FoV+* are 0.97, 0.98, 0.7 and 0.84, respectively. *FoV-only* is very vulnerable to prediction error because only predicted FoV content is streamed. Even though *FoV+* includes extra data to improve robustness against prediction error, it is still insufficient under large prediction error. In comparison, *Rubiks* does not incur noticeable video quality degradation: its difference from YouTube is only 0.01. This is because it streams the entire video frames albeit at a lower quality for tiles outside predicted FoV.

Video Quality Changes: Fig. 10(c) shows the video quality changes is 0.15 and 0.28 for *FoV-only* and *FoV+*. The average quality changes is around 0.01 for both *Rubiks* and YouTube. *FoV-only* and *FoV+* have higher video quality changes whenever the prediction error becomes large (*e.g.*, the user looks at video portions that lie outside the streamed content which leads to a large drop in viewing quality). Sending tiles not included in predicted FoV at lower quality allows *Rubiks* to avoid such a large drop in video quality.

Bandwidth Savings: Fig. 10(e) shows that for 4K videos *Rubiks* saves 35% bandwidth compared with YouTube. Among them, 19% bandwidth saving comes from *Rubiks* not sending all layers for all tiles, 11% saving comes from *Rubiks* sending tiles outside the predicted FoV at a lower rate, and 5% bandwidth saving comes from our removal of I-frame in the other layers introduced in Section 4.1. *FoV-only* and *FoV+* save 56% and 41% bandwidth compared with YouTube, respectively. The bandwidth saving of *Rubiks* is significant, but lower than *FoV* and *FoV+* since it streams entire video frames to improve robustness to movement prediction error. We believe this is a reasonable trade-off.

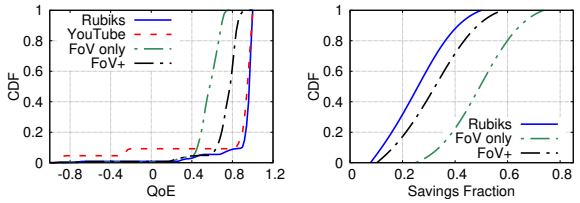
Low Throughput: Fig. 11 shows the benefits of *Rubiks* when throughput is lower than the lowest video bitrate. All approaches tend to select the lowest bitrate. YouTube has to send entire 360° frames, which results in larger rebuffering. The average QoE is 0.39, −0.52, 0.31 and 0.29 for *Rubiks*, YouTube, *FoV-only* and *FoV+*, respectively. Compared with YouTube, *Rubiks* improves QoE by 0.9 due to significant reduction in rebuffering time. Even if YouTube can support real-time decoding for 4K videos, reducing the amount of data sent provides significant benefits for *Rubiks* when network throughput is low. The average bandwidth savings of *Rubiks*, *FoV*, and *FoV+* are 39%, 61% and 48%, respectively.



(a) QoE (b) Bandwidth Savings

Figure 11: Performance of *Rubiks* under low throughput.

High-Motion Videos: Fig. 12 shows the benefits of *Rubiks* for high motion videos, which are encoded with larger size. When the user views high-motion tiles, there will be less chance to save bandwidth. The average bandwidth savings of *Rubiks* is 24%. *Rubiks* improves QoE by 16% on average over YouTube.

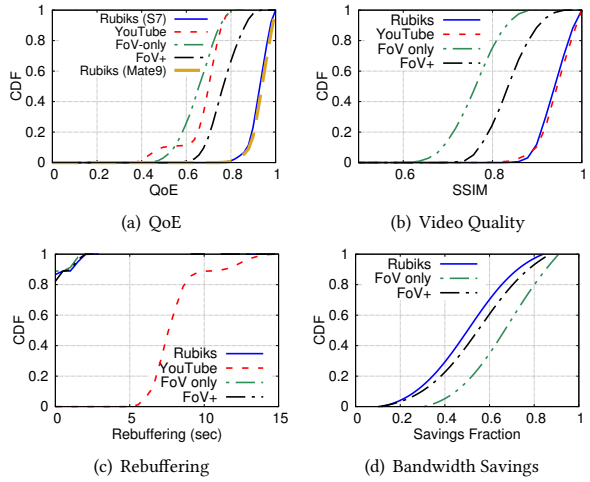


(a) QoE (b) Bandwidth Savings

Figure 12: Performance of *Rubiks* for high motion videos.

6.3.2 Rubiks *for 8K Videos.*

Next, we evaluate *Rubiks* for 8K videos. YouTube can not decode 8K video chunks timely. For *Rubiks* and *FoV+*, we use an upper bound, derived from our decoding time model, to limit the number of tiles sent to the client such that video chunks can be decoded before the playback deadline.



(a) QoE (b) Video Quality

(c) Rebuffering (d) Bandwidth Savings

Figure 13: Performance of 8K videos(8 videos, 10 throughput traces, 80 head movement traces)

QoE : Fig. 13(a) shows that *Rubiks* always achieves the best user QoE. *Rubiks* out-performs YouTube, *FoV-only* and *FoV+* by 36%, 40% and 20% in QoE, respectively. Compared with 4K videos, *Rubiks* achieves less QoE improvement over *FoV-only* for 8K videos due to different video content and user head movement. Because YouTube can not decode 8K video data in time, *Rubiks* achieves more QoE improvement over YouTube for 8K videos than that for 4K videos.

The QoE of *Rubiks* on Huawei Mate9 is 2% higher than that on Samsung S7. From the decoding time model, we can see that Mate9 has slightly faster decoding speed than S7. Thus, Mate9 can decode more tiles to provide higher robustness to head movement prediction error, which results in slightly higher QoE for Mate9.

Rebuffering: Fig. 13(c) shows the rebuffering time. Overall average rebuffering time of YouTube is 8.0 sec. Slow video decoding in the YouTube approach results in 7.1 sec average rebuffering time. The rest comes from throughput fluctuation during downloading. This leads to large QoE degradation. *Rubiks*, *FoV* and *FoV+* incur average rebuffering time in range 0.2-0.3 sec, which is very small when compared to YouTube. About 0.01 sec of average rebuffering time for *Rubiks* comes from inaccurate decoding time modeling. Compared with 4K videos, speeding up video decoding helps *Rubiks* achieve a larger reduction in rebuffering, which translates to a higher QoE improvement over YouTube.

Quality and Quality Changes: We observe similar video quality and video quality changes patterns as 4K videos. The average video quality is 0.96, 0.98, 0.79 and 0.87 for *Rubiks*, YouTube, *FoV-only* and *FoV+*, respectively. Compared with 4K videos, *FoV-only* achieves higher video quality due to slightly better head movement prediction. Nevertheless, *Rubiks* still significantly out-performs both *FoV-only* and *FoV+*. Moreover, *FoV-only* and *FoV+* experience 0.21 and 0.13 quality changes, which is larger than the other approaches due to movement prediction error, whereas, for *Rubiks*, it is 0.02.

Bandwidth Savings: Fig. 13(d) shows that *Rubiks*, *FoV-only* and *FoV+* save 49%, 66%, and 54% bandwidth, respectively, compared with YouTube. The encoded file size difference between two consecutive bitrates for 8K videos is larger than that in 4K videos. This means when we switch between bitrates in 8K videos, there is a larger difference in the amount of data when compared to 4K. So *Rubiks* yields larger bandwidth savings for 8K videos.

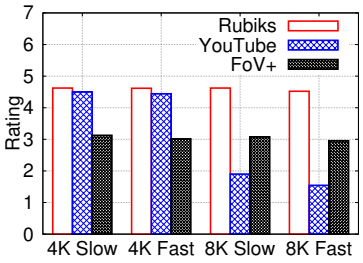


Figure 14: User Rating.

6.3.3 User Study. To evaluate whether our QoE metric can capture actual user experience, we conduct extensive user study. In our experiments, we let users view videos when running different algorithms and ask them to rate the quality with scores in range 1-5, where 1 corresponds to worst quality and 5 for best. We compare algorithms YouTube, *FoV+* and *Rubiks* but omit *FoV-only* due to its low QoE. We randomly shuffle the order of algorithms shown to the user for each experiment. We select two 4K videos and two 8K videos, one with fast motion and one with slow. For each video and algorithm, we run each experiment 3 times. 30 students participate in our user study, including 12 females and 18 males. Their ages range from 20 to 35. User watches the video in a standing position while wearing a mobile VR headset. Video data is transmitted to a smartphone through WiFi. We select one throughput trace whose average value is close to the median video bitrate.

Fig. 14 shows the average user rating for each video category. For 4K videos, the average rating is 4.6 and 4.5 for *Rubiks* and YouTube, respectively. Both algorithms achieve an average video SSIM of 0.98. *Rubiks* has no rebuffering, but YouTube has 0.2 sec rebuffering on average, which results in slightly lower rating. For 8K videos, *Rubiks* gets an average rating of 4.5, while YouTube gets 1.8, much lower than *Rubiks*, due to large rebuffering resulting from slow decoding. The average rebuffering for *Rubiks* and YouTube is 0.1 sec and 7.3 sec, respectively. The average rating of *FoV+* is 3.0 due to blank areas resulting from head movement prediction error. The average video SSIM for *FoV+* is 0.80, while that for other algorithms is 0.97. These results demonstrate that *Rubiks* achieves similar video quality for all video categories and significant QoE improvement for 8K videos compared with YouTube. Moreover, *Rubiks* has obvious QoE improvement from *FoV+* due to robustness to head movement prediction error.

6.3.4 Energy Consumption. We use the Google battery historian tool [21] to monitor the energy consumption of our system. We stream multiple 4K and 8K videos using various algorithms and record the energy consumption in each experiment, where each video lasts for 5 minutes. Table 2 shows the average total energy consumption across videos for each algorithm. Both *FoV-only* and *FoV+* consume up to 33% less energy than YouTube and *Rubiks* since they decode fewer tiles at the expense of much worse video quality. On average, *Rubiks* consumes 9% and 19% less energy than YouTube for 4K and 8K videos, respectively. This is because *Rubiks* does not decode all video tiles and takes shorter time to finish decoding than YouTube. When the decoder finishes decoding one chunk, it will

go to the idle state before the next chunk is ready to decode. As we would expect, all algorithms consume more energy in decoding the 8K video than the 4K video.

	YouTube	*Rubiks*	*FoV-only*	*FoV+*
4K(S7)	43.4mAh	39.8mAh	30.3mAh	34.5mAh
4K(Mate9)	41.9mAh	37.6mAh	29.2mAh	32.1mAh
8K(S7)	93.7mAh	76.2mAh	62.4mAh	68.6mAh
8K(Mate9)	91.8mAh	75.5mAh	61.7mAh	65.2mAh

Table 2: Energy Consumption.

6.4 Summary and Discussion of Results

Our main findings can be summarized as follow:

- *Rubiks* achieves significant QoE improvement due to reduction in rebuffering time and enhanced robustness against head movement prediction error.
- *Rubiks* saves substantial bandwidth by sending video tiles with a lower viewing probability at a lower quality.
- Users give higher ratings to *Rubiks* due to smaller rebuffering or higher video quality.

Though we evaluate *Rubiks*'s performance over a few smartphones, techniques employed in *Rubiks* can improve the performance of streaming 360° videos on any smartphone hardware platform. In addition to speeding up the decoding process, it gives the following benefits:

- *Rubiks* is more robust to head movement prediction errors even if decoding is fast. Compared with the existing state-of-art tile based streaming approaches *FoV-only* and *FoV+*, *Rubiks* improves video quality by 38% and 15% for 4K videos. The improvement is 22% and 10% for 8K videos. Note that, both *FoV-only* and *FoV+* can decode all 4K and 8K videos in real time. Thus, even if decoding is fast, the current tile based streaming approaches are not robust enough to head movement prediction errors.
- *Rubiks* needs much less bandwidth to achieve high video quality. Compared with YouTube, *Rubiks* saves 35% bandwidth for 4K videos, while achieving similar average video quality (0.97 in *Rubiks* and 0.98 in YouTube).

7 RELATED WORK

Video Streaming: There has been lots of recent works on video streaming under limited and fluctuating network throughput [22, 30, 32, 33, 36, 37, 47]. These works try to maximize the user QoE, which can be defined using multiple metrics: video bitrate, bitrate changes between successive chunks and rebuffering. Yin et al. [47] propose an MPC-based optimization framework for video streaming. It casts the problem as an utility optimization, where the utility is defined as the weighted sum of the above metrics for the future K video chunks. FESTIVE [32] balances both stability and efficiency, and provides fairness among video players by performing randomized chunk scheduling and bitrate selection of future chunks. Pensieve [36] trains a neural network that selects the bitrate of future chunks based on the performance data collected from video players. However, none of these works consider 360° video content streaming.

360° video Streaming: Recently, there have been some works targeting 360° video content streaming. In 360° videos, a user looks at only some portion of the video at any given time so there is an opportunity to save bandwidth without sacrificing quality. Qian et al. [39] propose a scheme that divides an entire 360° frame into several smaller rectangular tiles and only streams the tiles that overlap with predicted FoV. This approach can lead to rebuffering or blank screen in case of inaccurate head movement prediction. Hosseini et al. [29] propose an approach where the video is divided into multiple tiles and the tiles that are more likely to be viewed are streamed earlier. Bao et al. [23] propose an optimization framework that takes into account the head movement prediction error and requests some additional tiles to account for the prediction error. However, none of these approaches were implemented on smartphones. So it is not clear about the feasibility and performance of these existing approaches on smartphones. POI360 [46] proposes adapting compression ratio of video tiles according to network throughput, but it still suffers from slow decoding since the encoded rate of tiles does not affect decoding time. Moreover, POI360 is not implemented using video codecs available on commercial smartphones. Recently, Liu et al. [34] propose using SVC for 360° videos. However, SVC is currently not supported on smartphones, so they do not have a real implementation.

Video Encoding Schemes: There have been many works (*e.g.*, [25, 28, 31, 35, 41, 44]) on video encoding where some parts of the video are encoded at a higher bitrate, commonly referred to as Region of Interest (ROI) while other parts are encoded at a lower bitrate. This is only done for regular videos to account for the fact that some regions of the video contain more critical or useful information and should be encoded at a higher bitrate. This encoding is not suitable for 360° since the user can change FoV. So this does not scale because one has to handle large number of possible ROIs. Some works [27, 38, 40] try to use SVC to encode ROI with high quality. These works focus on optimizing user experience by exploiting SVC to reduce transmission delay, increase region of interest quality and avoid rebuffering.

Our work is inspired by these works, but differs from them in that our work targets specifically 360° and incorporates tile-based coding with layered coding to achieve efficient decoding, bandwidth saving, and robustness against prediction error.

8 CONCLUSION

In this work, we develop a novel tile-based layered streaming framework to tackle practical limitations of streaming 360° videos to smartphones. *Rubiks* divides 360° video chunks into spatial tiles and temporal layers. The layered design (i) enables real-time 360° video decoding by managing the number of layers sent for different tiles, (ii) accommodates head movement prediction error by streaming entire 360° frames and sending different portions at different quality, (iii) saves bandwidth by streaming the portion in the FoV at a higher quality while streaming the other parts at lower quality, and (iv) optimizes video quality by searching for an appropriate bitrate and the number of tiles given the predicted user's head movement and network condition. Through trace-driven system experiments and user study, we demonstrate the effectiveness of *Rubiks*. Moving forward, we plan to improve head movement prediction and also develop live 360° streaming service, which is more sensitive to transmission delay.

REFERENCES

[1] 2017. 360-Degree Football Game Video. (2017). https://www.youtube.com/watch?v=E0HUVPM_A00

[2] 2017. 360-Degree Rollercoaster Video. (2017). https://www.youtube.com/watch?v=8lsB-P8nGSM

[3] 2017. 360-Degree Sailing Video. (2017). https://www.youtube.com/watch?v=IJ_CwOFTZyM

[4] 2017. 360-Degree Shark Encounter Video. (2017). https://www.youtube.com/watch?v=rG4jSz_2HDY&t=15s

[5] 2017. Android Supported Media Formats. (2017). https://developer.android.com/guide/topics/media/media-formats.html

[6] 2017. Cisco Visual Networking Index Report. (2017). http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html

[7] 2017. Facebook Cubemap for 360 Degree Videos. (2017). https://code.facebook.com/posts/1126354007399553/next-generation-video-encoding-techniques-for-360-video-and-vr/

[8] 2017. Google Cardboard. (2017). https://store.google.com/us/product/google_cardboard

[9] 2017. H264. (2017). https://www.itu.int/rec/T-REC-H.264

[10] 2017. HEVC. (2017). https://www.itu.int/rec/T-REC-H.265

[11] 2017. HEVC Transform and Quantization. (2017). https://link.springer.com/chapter/10.1007/978-3-319-06895-4_6

[12] 2017. HSDPA TCP dataset. (2017). http://home.ifi.uio.no/paalh/dataset/hsdpa-tcp-logs/

[13] 2017. HTC Vive. (2017). https://www.vive.com

[14] 2017. Kvazaar. (2017). https://github.com/ultravideo/kvazaar

[15] 2017. LINUX tc. (2017). https://linux.die.net/man/8/tc

[16] 2017. Oculus. (2017). https://www.oculus.com

[17] 2017. Samsung Gear VR. (2017). http://www.samsung.com/us/mobile/virtual-reality/gear-vr

[18] 2017. Video Codec Hardware Acceleration. (2017). https://trac.ffmpeg.org/wiki/HWAccelIntro

[19] 2017. VR/AR Market. (2017). http://www.digi-capital.com/news/2017/01/after-mixed-year-mobile-ar-to-drive-108-billion-vrar-market-by-2021

[20] 2017. YouTube Encoder Settings for 360 Degree Videos. (2017). https://support.google.com/youtube/answer/6396222?hl=en

[21] 2018. Google Battery Historian Tool. (2018). https://github.com/google/battery-historian

[22] Saamer Akhshabi, Lakshmi Anantakrishnan, Constantine Dovrolis, and Ali C. Begen. 2013. Server-based Traffic Shaping for Stabilizing Oscillating Adaptive Streaming Players. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '13)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/2460782.2460786

[23] Y. Bao, H. Wu, T. Zhang, A. A. Ramli, and X. Liu. 2016. Shooting a moving target: Motion-prediction-based transmission for 360-degree videos. In *2016 IEEE International Conference on Big Data (Big Data)*. 1161–1170. https://doi.org/10.1109/BigData.2016.7840720

[24] A. Bjelopera and S. GrgiÄĞ. 2012. Scalable video coding extension of H.264/AVC. In *Proceedings ELMAR-2012*. 7–12.

[25] H. Chen, G. Braeckman, S. M. Satti, P. Schelkens, and A. Munteanu. 2013. HEVC-based video coding with lossless region of interest for telemedicine applications. In *2013 20th International Conference on Systems, Signals and Image Processing (IWSSIP)*. 129–132. https://doi.org/10.1109/IWSSIP.2013.6623470

[26] Mario Graf, Christian Timmerer, and Christopher Mueller. 2017. Towards Bandwidth Efficient Adaptive Streaming of Omnidirectional Video over HTTP: Design, Implementation, and Evaluation. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM, 261–271.

[27] D. Grois, E. Kaminsky, and O. Hadar. 2010. ROI adaptive scalable video coding for limited bandwidth wireless networks. In *2010 IFIP Wireless Days*. 1–5. https://doi.org/10.1109/WD.2010.5657709

[28] I. Himawan, W. Song, and D. Tjondronegoro. 2012. Impact of Region-of-Interest Video Coding on Perceived Quality in Mobile Video. In *2012 IEEE International Conference on Multimedia and Expo*. 79–84. https://doi.org/10.1109/ICME.2012.126

[29] Mohammad Hosseini and Viswanathan Swaminathan. 2016. Adaptive 360 VR Video Streaming: Divide and Conquer! *CoRR* abs/1609.08729 (2016). http://arxiv.org/abs/1609.08729

[30] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. 2014. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 187–198. https://doi.org/10.1145/2619239.2626296

[31] A. Jerbi, Jian Wang, and S. Shirani. 2005. Error-resilient region-of-interest video coding. *IEEE Transactions on Circuits and Systems for Video Technology* 15, 9 (Sept 2005), 1175–1181. https://doi.org/10.1109/TCSVT.2005.852619

[32] Junchen Jiang, Vyas Sekar, and Hui Zhang. 2012. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2413176.2413189

[33] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. 2014. Probe and Adapt: Rate Adaptation for HTTP Video Streaming At Scale. *IEEE Journal on Selected Areas in Communications* 32, 4 (April 2014), 719–733. https://doi.org/10.1109/JSAC.2014.140405

[34] Xing Liu, Qingyang Xiao, Vijay Gopalakrishnan, Bo Han, Feng Qian, and Matteo Varvello. 2017. 360Âř Innovations for Panoramic Video Streaming. In *Proc. of HotNets*. 50–56.

[35] Y. Liu, Z. G. Li, and Y. C. Soh. 2008. Region-of-Interest Based Resource Allocation for Conversational Video Communication of H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology* 18, 1 (Jan 2008), 134–139. https://doi.org/10.1109/TCSVT.2007.913754

[36] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 197–210. https://doi.org/10.1145/3098822.3098843

[37] C. Mueller, S. Lederer, J. Poecher, and Ch. Timmerer. 2013. libdash - An Open Source Software Library for the MPEG-DASH Standard. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME) 2013, San Jose, USA*. pp. 1–2.

[38] J. R. Ohm. 2005. Advances in Scalable Video Coding. *Proc. IEEE* 93, 1 (Jan 2005), 42–56. https://doi.org/10.1109/JPROC.2004.839611

[39] Feng Qian, Lusheng Ji, Bo Han, and Vijay Gopalakrishnan. 2016. Optimizing 360 Video Delivery over Cellular Networks. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges (ATC '16)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2980055.2980056

[40] T. Schierl, T. Stockhammer, and T. Wiegand. 2007. Mobile Video Transmission Using Scalable Video Coding. *IEEE Transactions on Circuits and Systems for Video Technology* 17, 9 (Sept 2007), 1204–1217. https://doi.org/10.1109/TCSVT.2007.905528

[41] P. Sivanantharasa, W. A. C. Fernando, and H. K. Arachchi. 2006. Region of Interest Video Coding with Flexible Macroblock Ordering. In *First International Conference on Industrial and Information Systems*. 596–599. https://doi.org/10.1109/ICIIS.2006.365798

[42] Thomas Stockhammer. 2011. Dynamic adaptive streaming over HTTP–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*. ACM, 133–144.

[43] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. 2016. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 272–285.

[44] N. Tsapatsoulis, C. Loizou, and C. Pattichis. 2007. Region of Interest Video Coding for Low bit-rate Transmission of Carotid Ultrasound Videos over 3G Wireless Networks. In *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. 3717–3720. https://doi.org/10.1109/IEMBS.2007.4353139

[45] Zhou Wang, Ligang Lu, and Alan C Bovik. 2004. Video quality assessment based on structural distortion measurement. *Signal processing: Image communication* 19, 2 (2004), 121–132.

[46] Xiufeng Xie and Xinyu Zhang. 2017. POI360: Panoramic Mobile Video Telephony over LTE Cellular Networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 336–349.

[47] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 325–338. https://doi.org/10.1145/2785956.2787486

[48] Chao Zhou, Zhenhua Li, and Yao Liu. 2017. A measurement study of oculus 360 degree video streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM, 27–37.