Flare: Practical Viewport-Adaptive 360-Degree Video Streaming for Mobile Devices

Feng Qian[1]* Bo Han[2] Qingyang Xiao[1] Vijay Gopalakrishnan[2]

[1]Indiana University [2]AT&T Labs – Research

ACM Reference Format:
Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. 2018. Flare: Practical Viewport-Adaptive 360-Degree Video Streaming for Mobile Devices. In *MobiCom '18: 24th Annual Int'l Conf. on Mobile Computing and Networking, Oct. 29–Nov. 2, 2018, New Delhi, India.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

Abstract

Flare is a practical system for streaming 360° videos on commodity mobile devices. It takes a viewport-adaptive approach, which fetches only portions of a panoramic scene that cover what a viewer is about to perceive. We conduct an IRB-approved user study where we collect head movement traces from 130 diverse users to gain insights on how to design the viewport prediction mechanism for Flare. We then develop novel online algorithms that determine which spatial portions to fetch and their corresponding qualities. We also innovate other components in the streaming pipeline such as decoding and server-side transmission. Through extensive evaluations ($\sim$400 hours' playback on WiFi and $\sim$100 hours over LTE), we demonstrate that Flare significantly improves the QoE in real-world settings. Compared to non-viewport-adaptive approaches, Flare yields up to 18× quality level improvement on WiFi, and achieves considerable bandwidth reduction (up to 35%) and video quality enhancement (up to 4.9×) on LTE.

1 INTRODUCTION

360° video, *a.k.a.* panoramic or immersive video, is becoming popular on commercial video platforms such as YouTube and Facebook [10, 13]. It brings an immersive experience to users by projecting the panoramic content onto the display. The user's viewport (visible area) is determined in real time by her viewing direction and the Field of View (FoV) of the headset, and changes as she moves her head.

The challenge with 360° videos, however, is that they are much larger (4× to 6×) than conventional videos under the same perceived quality due to their panoramic nature. Therefore, streaming 360° videos is challenging, especially over wireless (*e.g.,* cellular) networks where bandwidth can be scarce. In addition, streaming 360° videos incurs higher CPU, GPU, and energy overhead on the client side compared to the streaming of regular videos [37].

In this paper, we present the design, implementation, and evaluation of Flare, a practical 360° video streaming system for commodity mobile devices. Flare adopts a *viewport-adaptive* approach: instead of downloading the entire panoramic scene, it predicts the user's future viewports and fetches only portions that cover what the viewer is about to consume. As a result, Flare can significantly reduce the bandwidth usage, or boost the video quality given the same bandwidth compared to the state-of-the-art. Moreover, Flare is a *general* 360° video streaming framework that does not depend on a specific video encoding technology.

The high-level principle of viewport-adaptive 360° streaming is not new [23, 48]. However, there are indeed numerous technical challenges that need to be addressed. To the best of our knowledge, Flare is one of the first full-fledged systems that realize this principle on commodity smartphones. The key design aspects of Flare consist of the following.

First, an essential component in Flare is to predict users' future viewports (*e.g.,* via head movement prediction). We systematically investigate real users' head movement as well as how to efficiently perform viewport prediction (VP) on mobile devices. We conduct an IRB-approved user study involving 130 diverse users in terms of age, gender, and experience. To the best of our knowledge, this is the 360° video head movement dataset with the longest viewing time the research community has ever collected. We intend to make the dataset publicly available. Using this dataset consisting of 4420-minute 360° video playback time, we study a wide spectrum of Machine Learning (ML) algorithms for VP. We then design lightweight but robust VP methods for Flare by strategically leveraging off-the-shelf ML algorithms (§3).

Second, a prerequisite for viewport-adaptive streaming is to spatially segment 360° video contents into smaller portions

[*] Current affiliation: University of Minnesota – Twin Cities.

MobiCom'18, October 29–November 2, 2018, New Delhi, India
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5903-0/18/10…$15.00
https://doi.org/10.1145/XXXXXX.XXXXXX

called *tiles*. We evaluate the overhead incurred by re-encoding a set of 360° videos, and find out that for most videos we tested, the overhead is acceptable – typically around 10% of the original video size. This indicates that from the encoding perspective, tile-based streaming is feasible and potentially beneficial for most 360° videos (§4).

Third, Flare needs to make decisions in both the spatial domain (which tiles to fetch) and the quality domain (which qualities to fetch), both jointly forming a huge search space. Doing so is in particular challenging in the face of imperfect VP and network capacity estimation. We decouple the two dimensions by first determining the to-be-fetched tiles based on the VP – oftentimes we need to strategically fetch more tiles than those predicted to tolerate the VP inaccuracy (§4). We then formulate an optimization algorithm to determine the tile quality. The algorithm takes into account multiple conflicting objectives of maximizing the tiles' bitrate, minimizing the stalls, and minimizing the cross-tile quality differences. We also conduct meticulous optimizations for our rate adaptation scheme to accelerate it at runtime in order to adapt to users' fast-paced head movement when needed (§5).

Fourth, we bring a number of innovations when integrating the above components into a holistic system. Our player follows the Dynamic Adaptive Streaming over HTTP (DASH) paradigm with all key logic on the client side. Specifically, Flare simultaneously leverages multiple hardware decoders that are available on modern smartphones. It introduces a cache in the video memory that stores decoded tiles, in order to facilitate smooth viewport transitions as the displayed tiles change (§6). We also consider the non-trivial decoding delay in the rate adaptation formulation (§7.2). On the server side, despite its stateless logic, we enhance its networking subsystem by making it adaptive to the continuous tile request stream as the user changes her viewing direction (§7.1, §8).

We have implemented Flare on Android smartphones and Linux servers in 14,200 lines of code. We conduct extensive evaluations via real-world experiments over WiFi through more than 400 hours' playback. We also evaluate Flare on commercial cellular networks at 9 locations in 6 states in the U.S. with ~100 hours' video playback. We highlight our key evaluation results below.

- On networks with constrained bandwidth (emulated over WiFi), Flare improves the video quality by a factor between 1.9× and 18×, compared to non-viewport-adaptive schemes. The average stall of Flare is less than 1 second per minute.
- On real LTE networks, Flare simultaneously achieves bandwidth reduction (from 26% to 35%), quality level improvement (around 22%), and comparable stall duration, compared to non-viewport-adaptive schemes. In locations with poor cellular conditions, Flare can improve the quality level by up to 4.9× while reducing the stall time by 22%.

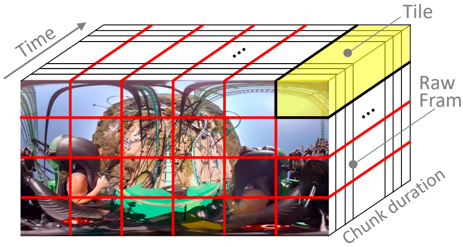

Figure 1: **Chunk, tile, frame, and tile segmentation (4×6 tiles).**

- Flare's various system-level optimizations are highly effective. For example, compared to using a single H.264 decoder, employing multiple decoders reduces the stall duration by 90%. Also, our server-side transport-layer modification reduces the stall by 39%.
- We conduct best-effort implementations of other tile-based streaming algorithms in recent proposals [29, 47]. Compared to them, Flare achieves significantly better performance such as up to 98.7% reduction of stall time over LTE.

Overall, we demonstrate that it is entirely feasible to develop a viewport-adaptive 360° video streaming system on commodity mobile devices without relying on specialized infrastructure or modifications on video encoding. We summarize our key contributions as follows.

- A large-scale user study with diverse users and 360° video contents, to understand real users' head movement dynamics.
- The design and implementation of Flare, which consists of a novel framework of the end-to-end streaming pipeline. Flare introduces new streaming algorithms, and brings numerous system-level and network-level optimizations. We manage to integrate all design components into a holistic mobile system running on commodity smartphones.
- Extensive evaluations of Flare in real WiFi and cellular networks with diverse conditions, and comparison with other viewport-adaptive and non-viewport-aware approaches.

A demo video of Flare can be found at: https://goo.gl/huJkXT

2 MOTIVATION AND FLARE OVERVIEW

Almost all commercial 360° video content providers today employ a monolithic streaming approach that fetches the entire panoramic content regardless of the user's viewport [21, 48]. This approach is simple but causes considerable waste of the network bandwidth, as a user consumes only a small portion of the panoramic scene in her viewport. To overcome this limitation, several recent studies propose to make 360° video streaming *viewport-adaptive* by fetching only content in the predicted viewport, or fetching in-viewport content at a higher quality compared to non-viewport content. Within different viewport-adaptive approaches [23, 26, 42, 47, 48, 52, 62], we adopt the *tile-based* solution [42, 48] due to its conceptual simplicity and potential benefits. As illustrated in Figure 1, an original video chunk is segmented into *tiles*. A tile (*i.e.,* the yellow area) has the same duration and number

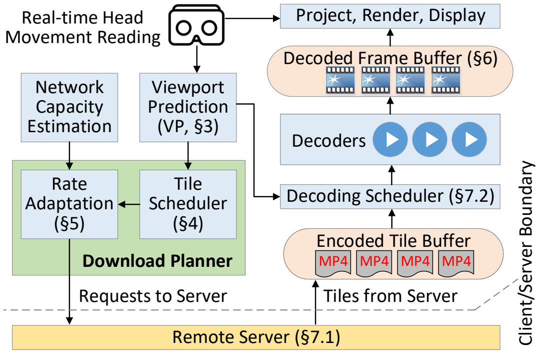

Figure 2: The Flare system.

of frames as the chunk it belongs to, but occupies only a small spatial portion. Each tile can be independently downloaded and decoded. Therefore, ideally a player needs to download only tiles that cover a user's viewport trajectory.

Flare Overview. The high-level idea behind Flare is intuitive: the player predicts a user's head movement (*i.e.,* viewport) and prefetches the tiles to be consumed by the user. Despite this intuitive idea, there remain numerous challenges in designing such a system. First, Flare should be *highly responsive* to fast-paced viewport changes and viewport prediction (VP) updates. Second, targeting the ABR (adaptive bitrate) streaming, Flare needs a practical and effective rate adaptation algorithm that determines the qualities of tiles by considering both the network capacity and head movement, yielding a potentially huge search space. Third, Flare is designed for off-the-shelf mobile devices whose processing capabilities are much weaker than PCs. In particular, the total time budget for the overall processing pipeline, which is performed entirely on the client, is typically less than 1 second as limited by the time window that can yield a reasonable VP accuracy (§3.2).

Figure 2 sketches the system design of Flare. The client performs VP in real time (§3). Then a very important component on the client side is the *Download Planner*. It takes as streamed input the VP and network capacity estimation, and computes the set of tiles to be downloaded (§4) as well as their desired qualities (§5). When tiles arrive from the server, they are properly buffered, decoded, projected, and rendered to the viewer, as shown in the RHS of Figure 2. We describe this process in §6 and §7.2. Compared to the client side, the server is relatively "dumb" – simply transmitting the tiles per clients' requests. This client-server function partition follows the DASH streaming paradigm, which facilitates scalability and ease of deployment, as to be detailed in §7.1.

3 VIEWPORT PREDICTION

3.1 User Study

To understand how real users watch 360° videos, we conduct an IRB-approved user study involving 130 subjects.

Dataset	Videos		Users	Views		Sample
	#	Len	#	#	Len	Freq.
Qian *et al.* [48]	4	10min	5	20	50min	250Hz
Corbillon *et al.* [25]	5	6min	59	295	354min	30Hz
Lo *et al.* [43]	10	10min	50	500	500min	30Hz
Bao *et al.* [23]	16	8min	153	985	492min	7-9Hz
Wu *et al.* [55]	18	44min	48	864	2112min	100Hz
Our dataset	10	34min	130	1300	4420min	200Hz

Table 1: Comparing our dataset with existing ones.

Video Selection. We select ten popular 360° videos[1] from YouTube (each having at least 2M views as in March 2018). The videos span a wide range of genres containing both low-motion and high-motion scenes. They are all in 4K resolution, using Equirectangular projection, and encoded in standard H.264 format. Their bitrates range from 12.9 to 21.5 Mbps, and their playback duration ranges from 117s to 293s.

Participant Recruitment. 133 voluntary subjects, with ages ranging from 18 to 68 years old, participated in our user study and were asked to watch all videos. 100 subjects were students, staff, and faculty members from a large university in the U.S., and the remaining users were from a large company. 42% of them are female and 48% of them have never seen 360° videos before. Regarding the age distribution, the numbers of subjects who are teenagers, 20s, 30s, 40s, 50s, and 60s are 5, 79, 27, 13, 2, and 4, respectively. 35% of the subjects are older than 30. Each subject receives a small amount of compensation for his/her participation.

Data Collection. Each subject watches the ten videos in an arbitrary order by wearing a Samsung Gear VR headset with a Samsung Galaxy S8 (SGS8) smartphone plugged into it. The player collects head orientation data (pitch, yaw, and roll) from built-in sensors at a high sampling rate (∼200 Hz). When watching the videos, a user is free to turn her head around. A user may pause for an arbitrary amount of time or take off the VR headset after watching each video. Also, if a user experiences motion sickness, she may exit the study at any time. 3 such users did not finish the study. They are not counted into the 130 users.

Dataset. We process the head movement traces and create a dataset to be used in our study. We convert the raw data into the spherical coordinates (latitude and longitude[2]) using a downsampling rate of 100Hz in order to reduce the processing overhead and the possibility of overfitting in VP models. Further downsampling to 50Hz yields similar VP accuracy.

In total, we have 1,300 (130×10) views from the subjects and the total duration of our collected trace is 4,420 minutes. As shown in Table 1, compared to other head movement

[1] A list of the videos: wild animals [4], sci-fi [1], sea [2], roller coaster [7], island [11], racing [5], house [9], concert [19], tennis [16], skydiving [18].

[2] In this study we consider only the *pitch* and *yaw*, as users rarely change the *roll* (*i.e.,* rotating head along the Z axis) when watching VR contents [23].

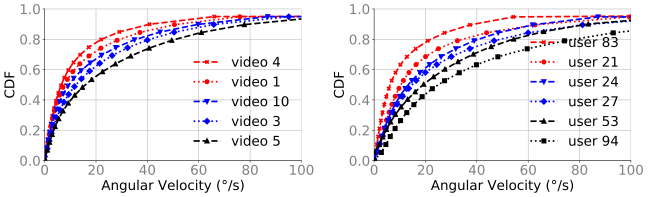

Figure 3: Angular velocity for sampled videos & users.

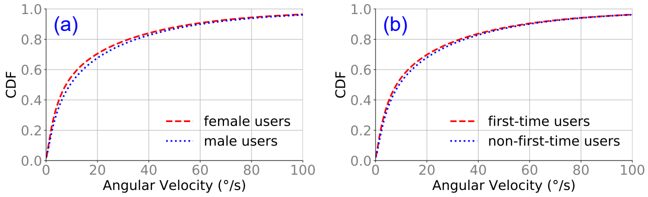

Figure 4: Comparing (a) female vs. male, and (b) first-time vs. non-first-time viewers.

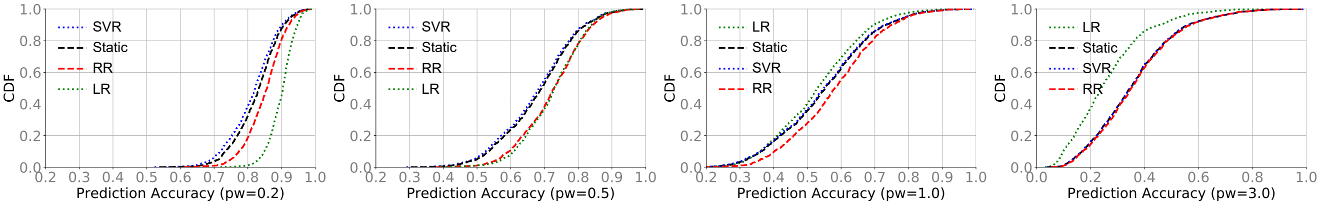

Figure 5: VP accuracy for four prediction window (pw) sizes: 0.2s, 0.5s, 1.0s, and 3.0s using four ML algorithms, assuming 4×6 tiles.

datasets [23, 25, 43, 55], ours is the most comprehensive and diverse dataset for 360° videos with the longest viewing time.

To demonstrate the diversity of our participants and videos, the two subplots in Figure 3 show the angular velocity distributions for randomly selected videos and users, respectively. As shown, the distributions differ noticeably across both videos and users. The mean angular velocity of the head rotation for playback sessions across all users and videos is 10.3°/sec with the standard deviation being 7.4°/sec. We also compare the head movement speed between first-time and non-first-time viewers, as well as between male and female subjects. As shown in Figure 4, we did not observe noticeable difference from their distributions of the angular speed, likely because watching 360° videos is quite intuitive and straightforward.

3.2 VP Method for Flare

At a high level, there are two opportunities that we can leverage for VP. First, we can use an individual user's historical head movement to predict her viewport in the *near* future (*e.g.,* 1 to 2 seconds) [23, 48]; second, we can possibly use multiple users' head movement trajectories to identify "popular" spatial regions to facilitate content prefetching in the *distant* future [42]. In Flare, we focus on the former and leave the crowd-source based prediction for future work.

Several prior studies indicate that VR users' head movement is indeed predictable [23, 48]. We follow a typical online Machine Learning (ML) paradigm by using the most recent hw (history window) seconds worth of head movement data to predict the viewport in the next pw (prediction window) seconds. This involves both training and testing. A design decision we need to make is to determine which ML algorithm to use: we can either use cheap but less accurate ML algorithms to perform frequent VP, or use more accurate ML algorithms that however may run slowly on commodity smartphones. In our design, we take the former approach with the purpose of adapting to the fast-paced head movement.

Following this idea, we consider four off-the-shelf ML algorithms: *Static*, *Linear Regression (LR)*, *Ridge Regression (RR)*, and *Support Vector Regression (SVR)*. *Static* simply uses the current head position to approximate one in the future without any "real" prediction; LR treats the viewport trajectory in pw as time series and estimates the future viewport using a linear model; RR is a more complex variant of LR and can better cope with overfitting [50]; SVR leverages Support Vector Machine (SVM) to perform regression [51]. For LR, RR, and SVR, the latitude and longitude trajectories are predicted separately. Also, instead of using a fixed hw, we find that dynamically adjusting the hw to be *proportional* to pw yields better VP accuracy. We therefore empirically set $hw = pw/2$ in our implementation (changing the coefficient does not qualitatively affect the results).

Prediction Results. We apply the above methods to all 1,300 head movement traces collected from our user study. The four plots in Figure 5 show the distributions of the VP accuracy for four pw sizes: 0.2s, 0.5s, 1.0s, and 3.0s, across all traces. We define a VP instance to be *accurate* if and only if $\Pi_{\text{viewed}} \subseteq \Pi_{\text{predicted}}$, where Π_{viewed} is the set of tiles actually perceived by the viewer (the ground truth), and $\Pi_{\text{predicted}}$ is the tile set determined based on the predicted lat/lon, assuming a 4×6 tile segmentation. In Figure 5, each data point on a CDF curve is the VP accuracy for one head movement trace (a user watching one video). The per-trace VP accuracy is defined as the number of accurate VPs divided by the total number of VPs performed. As shown, due to users' head movement randomness, accurate VP is only achievable for short pw. Also, different ML algorithms exhibit diverse performance for different pw. For example, LR performs quite well for short pw but its accuracy degrades when pw increases, likely due to overfitting that can be mitigated by RR. For the four pw sizes, the median VP accuracy achieved by the best of the four ML algorithms is 90.5%, 72.9%, 58.2%, and 35.2%, respectively. Based on the results in Figure 5, Flare uses LR for $pw < 1$ and RR for $pw \geq 1$ due to their lightweight nature and reasonable accuracy.

4 TILE SCHEDULER

Measuring Tile Segmentation Overhead. Before detailing the design of our tile scheduler, we first examine the tile segmentation procedure itself. The number of tiles incurs a tradeoff: since a tile is an atomic downloadable unit, having too few tiles (too coarse-grained segmentation) limits the bandwidth saving, while having too many tiles (too fine-grained segmentation) leads to larger video sizes due to the loss of cross-tile compression opportunities. We use the 10 videos in our user study to quantify such an overhead. For each video, we first preprocess it into the standard DASH format using a chunk duration of 4 seconds as recommended by prior work [34, 44, 59]. We then perform two lossless operations on the preprocessed videos: (1) further reducing the chunk duration to 1 second, and (2) tile segmentation. We need the first operation because a tile with a shorter duration offers more flexibility for tile-based streaming, but the negative side is increased chunk size due to denser I-frames (the first frame of each tile/chunk must be an I-frame).

Figure 6 shows the per-video segmentation overhead (measured as the fraction of increased bytes) for 1-second 4×6, 4×4, and 2×4 tiles, using the 4-second DASH chunks as the comparison baseline. As shown, the overhead ranges from about 5% to 15%, with the median across 10 videos being 7.3%, 9.3%, and 10.4% for 2×4, 4×4, and 4×6 tiles, respectively. These are acceptable because the benefits provided by tile-based streaming overweigh the incurred overhead, as to be demonstrated later. The segmentation overhead is taken into account by all evaluations conducted in this paper.

Download Planner Overview. As shown in Figure 2, the *Download Planner* is responsible for determining (1) which tiles are to be fetched, and (2) what their qualities should be. In theory, these two aspects need to be jointly considered. But a key design decision we make is to *separate these two decision processes*, *i.e.,* first calculating the to-be-fetched tiles (the job of the *tile scheduler*), and then determining their qualities (*rate adaptation*). The rationale behind this is two-fold. First, jointly considering both leads to a big inflation of the decision space and thus the algorithm complexity; second, (1) is more important than (2) because a missing tile will inevitably cause stalls. We describe the tile scheduler in the remainder of this section and the rate adaptation in §5.

Clearly, if the viewer's head movement is perfectly known beforehand, then the required tiles can be calculated deterministically. In reality, however, they can be only estimated from the imperfect VP. Since the head movement keeps changing, Flare needs to *continuously* perform VP.

Calculating the Tile Set. Assume that at time T_0, VP is invoked to update the to-be-fetched tile list. Instead of performing a single prediction, Flare conducts *multiple* predictions for time $t = T_0 + \delta_T$, $T_0 + 2\delta_T$, ..., $T_0 + m\delta_T$, in order to construct the *trajectory* of the user's future viewports. In other words, when invoked at T_0, the VP module outputs m tuples (t_i, ϕ_i, λ_i) each indicating the predicted lat/lon at time $t_i = T_0 + i\delta_T$. Note that as the prediction window (whose length is $m\delta_T$) moves forward, the same timestamp will be predicted multiple times at different VP invocations, as long as the timestamp is within the prediction window. The tile scheduler then translates each tuple into a tile set, whose chunk number is determined by $\lfloor t_i/d \rfloor$ (d is the chunk duration) and its tile numbers are determined by ϕ_i (predicted latitude), λ_i (predicted longitude), and the viewport-to-tile mappings (pre-generated based on the projection algorithm, as exemplified in Figure 7). Finally, the set of tiles to be fetched consists of the *union* of all predicted tile sets across all m predictions, excluding those already received. Regarding selecting m and δ_T, ideally m should be large and δ be small. Considering the system overhead and the difficulty of VP in the long term (§3.2), we pick δ_T=100ms and m=30.

Tolerating VP's Inaccuracy. The above procedure selects all tiles that fall into any viewport along the predicted trajectory. But doing so is not sufficient because VP may make errors. Flare employs three mechanisms to tolerate inaccurate VP. The first one is naturally provided by the tiles themselves: since a tile must be fetched as long as *any* of its frames intersects with *any* predicted viewport, oftentimes only a small portion of a fetched tile is consumed. This wastes some bandwidth but helps absorb inaccurate VP as long as the error does not cross a tile.

The second mechanism dealing with inaccurate VP is to fetch additional tiles that are not in the originally predicted tile set. We call them "out-of-sight" (OOS) tiles because if the prediction is accurate, a viewer will not see those tiles. We next describe the methodology of selecting OOS tiles. We assume that for each viewport in the predicted trajectory, all tiles (including those not in the viewport) are *ranked* by their "perceptive importance". Intuitively, tiles that fully appear inside the viewport are ranked the highest, followed by tiles that partially overlap with the viewport, then tiles near the viewport, and finally "far-away" tiles that are in the opposite direction of the viewport. We describe how to rank the tiles shortly. For a given viewport v, Flare fetches its k tiles with the highest ranks. k is calculated as follows:

$$k(v) = c_0(v) + \lceil \xi(1 - S)(n - c_0(v)) \rceil \qquad (1)$$

where $c_0(v)$ is the number of tiles that overlap with the viewport. $k(v)$ must be at least $c_0(v)$, otherwise the user's view

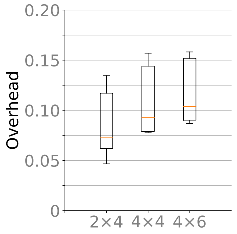

Figure 6: **Segmentation overhead.**

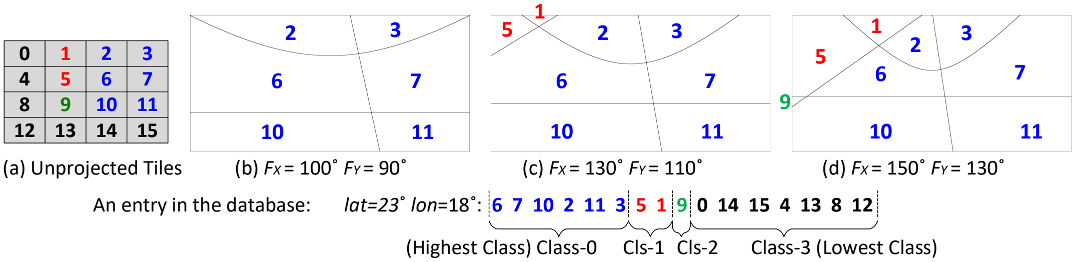

Figure 7: **Example: ranking the tiles and deriving classes for a viewport. Assume 4×4 tiles and 4 classes.**

becomes incomplete at v. For the remaining $n - c_0(v)$ OOS tiles where n is the total number of tiles per chunk, we fetch the top $\xi(1 - S)$ fraction of them, again based on the tiles' ranks. ξ is a parameter controlling the aggressiveness of fetching OOS tiles; $S \in [0, 1]$ quantifies the recent VP accuracy. Intuitively, the second part in Eq. 1 adjusts the number of OOS tiles to be fetched adaptively based on S: if the recent VP becomes accurate (inaccurate), then less (more) OOS tiles will be scheduled for fetching. S is computed as the Exponentially Weighted Moving Average (EWMA) of the VP accuracy defined as $J(\hat{L}, L)$. $J()$ is the Jaccard Index (a metric measuring set similarity [49]); L is the actually consumed tile set for the current frame, and $\hat{L}$ is its prediction conducted moments before (we use 200ms).

$$S \leftarrow \alpha \cdot J(\hat{L}, L) + (1 - \alpha) \cdot S \qquad (2)$$

The third mechanism tackling the imperfect VP is to let an earlier inaccurate prediction be fixed by a more recent prediction that is more accurate. We describe it in §7.1.

Ranking the Tiles. A viewport is defined as $(\phi, \lambda, F_x, F_y)$ where ϕ and λ are the latitude and longitude of the viewing direction, respectively, and F_x and F_y are the (constant) width and height of the viewport. We now describe given a viewport, how to rank all tiles based on their "perceptive importance". Recall the ranks are used in selecting OOS tiles, and it will also be used in rate adaptation. We begin with tiles that overlap with the viewport. We call them Class-0 tiles, and rank them according to their visible areas in the viewport. For example, Figure 7(a) plots a 4×4 tile segmentation configuration. Figure 7(b) shows the tiles that a user sees when looking at ϕ=23°, λ=18° with F_x=100° and F_y=90°. Tile 6 ranks the highest because it occupies the largest area, followed by Tile 7, and so on. To rank the remaining OOS tiles, we expand the FoV by increasing F_x and F_y (ϕ and λ remain the same). Then additional tiles (*e.g.,* Tiles 5 and 1 in Figure 7(c)) may become visible. We call these new tiles Class-1 tiles, and also rank them according to their areas in the extended viewport. We then use the same approach of extending the viewport to rank tiles in lower classes (Class-2 and so on). For all remaining tiles, they belong to the lowest class and are ranked by the spherical distance between their centroid and the center of the viewport. Intuitively, the above procedure creates a *total order relation* on the overall tile set for each viewport. Also, it is important to note that the whole ranking process is performed *offline* and cached in a database for runtime use, as exemplified in Figure 7.

Putting Everything Together. We now summarize the overall process of scheduling an ordered list of tiles to fetch. The following steps are performed on a per-frame basis to adapt to the viewer's continuous head movement. (1) Compute the Jaccard Index and use that to update S (Eq. 2) and henceforth $k(v)$ (Eq. 1). (2) Perform VP for m future timestamps $\{t_i = T_0 + i\delta_T\}$. (3) For each t_i, perform the database lookup using its predicted lat/lon as the key; select the ordered sublist containing the first $k(v)$ tiles from the database entry. (4) Merge the sublists across $\{t_i\}$ into the final (ordered) list containing the tiles to fetch. In the merged list, tiles are sorted by their predicted occurrence time (the primary key) and their ranks as calculated in Figure 7 (the secondary key). (5) Pass this merged list to the rate adaptation algorithm.

5 RATE ADAPTATION

In non-360° video streaming, numerous rate adaptation schemes have been proposed [34, 36, 44, 59]. Flare's rate adaptation approach is inspired by MPC [59], which provides a principled model that jointly considers different QoE objectives for Internet videos. Leveraging the high-level concept from MPC, we develop a practical formulation for rate adaptation of tile-based 360° video streaming.

Defining QoE Metrics. For DASH-style non-360° videos, commonly used QoE metrics consist of stall duration (the shorter the better), video quality (the higher the better), and inter-chunk quality switches (the fewer the better). Since we are not aware of any prior study that specifically focuses on 360° videos' QoE metrics, we propose the following metrics that our algorithm will optimize.

- The stall duration, denoted as T_{stall}. A stall happens when either the buffer is empty or a required tile is missing, in which case the player freezes the playback to fetch that tile.

• The average bitrate that is *actually consumed* by the viewer. It is computed as

$$B = \sum_{i,j} w(i,j)b(i,j)/\textit{VideoLength} \tag{3}$$

where $b(i,j)$ is the playback bitrate of Tile j of Chunk i, and $w(i,j) \in [0,1]$ is the fraction of the tile (averaged across all its frames) that fully or partially overlaps with the viewport.
• The average quality level (with 0 being the lowest) of all consumed tiles, defined as

$$Q = \sum_{i,j} \mathbf{1}(w(i,j))l(i,j)/\textit{VideoLength} \tag{4}$$

where $l(i,j)$ is the quality level of Tile j of Chunk i, and $\mathbf{1}(x)$=1 if and only if $x > 0$ otherwise $\mathbf{1}(x)$=0. We find this metric is largely correlated with B but is easier to compute.
• The quality switch consists of two parts: the *inter-chunk switch* and the *intra-chunk switch*. The former captures the quality switch between neighboring chunks. It is defined as the average change of the average consumed tiles' qualities between consecutive chunks:

$$I_1 = \frac{\sum_i \left| \frac{\sum_j \mathbf{1}(w(i,j))l(i,j)}{\sum_j \mathbf{1}(w(i,j))} - \frac{\sum_j \mathbf{1}(w(i-1,j))l(i-1,j)}{\sum_j \mathbf{1}(w(i-1,j))} \right|}{\textit{VideoLength}} \tag{5}$$

The intra-chunk switch quantifies the variation of qualities of consumed tiles belonging to the same chunk. It is defined as:

$$I_2 = \sum_i \textbf{StdDev}\left\{l(i,j)|\forall j : \mathbf{1}(w(i,j)) > 0\right\}/\textit{VideoLength} \tag{6}$$

Then the overall quality switch is calculated as a weighted sum of I_1 and I_2. We empirically set both weights to 1. Note that for 360° videos streamed using the conventional approach (one tile per chunk), I_2 is always 0.

The above metrics are either identical to or inherited from those for regular videos. Most of them are intuitive and easy to reason. The likely exceptions are inter- and intra-chunk switch, whose QoE impacts for 360° videos are not fully explored yet. As shown in Figure 7, even when the tiles have the same quality level, they will have different perceived qualities after projection that either enlarges or shrinks each tile. When the tiles' quality levels differ, their perceived qualities after projection become even more complex. We will study this in future work.

Problem Formulation. With the QoE metrics defined, the rate adaptation problem can be formulated as the following: determining the quality level for each to-be-fetched tile (determined in §4) so that the overall utility, defined as a weighted sum of the aforementioned metrics, is maximized:

$$\textbf{To Maximize: } \text{Utility} = Q - w_i(I_1 + I_2) - w_s T_{stall} \tag{7}$$

where w_i and w_s are weights that penalize quality switches and stalls, respectively.

We next convert the above formulation into its *online version*, with two design points being highlighted below. First, the online version only considers the tiles within a limited window of $m \cdot \delta_T$ seconds (§4). Second, the online version should deal with tiles with different viewing probabilities. Recall that oftentimes OOS tiles (with classes lower than 0) are scheduled for fetching. The OOS tiles have lower probabilities of being viewed compared to Class-0 tiles, and henceforth should be treated differently in the rate adaptation formulation. We adopt a simple approach of weighing tiles based on their classes. In Eq. 4, we replace $l(i,j)$ with $l(i,j)c(i,j)$ where $c(i,j)$ is the weight of the tile's class, defined as the highest class that the tile belongs to across all viewports in the predicted trajectory. The higher the class is, the higher its weight is. We empirically find the following weight function Class-$k \rightarrow 1/2^k$ works well in practice. For I_1 and I_2, due to the ways they are computed, we should not directly modify their $l(i,j)$ with $c(i,j)$. So we rewrite I_1 as the weighted sum of the *class-wise* inter-chunk quality switches with Class-k weighted by $1/2^k$; we rewrite I_2 as the weighted sum of intra-class quality switches for each class, plus the weighted sum of inter-class quality switches within each chunk.

Finding a Solution Fast. Having the online version of the formulation, we now consider how to solve it. The key challenge is to find a solution *fast*: due to the continuous head movement, the optimization should be invoked at a much higher frequency compared to conventional DASH algorithms. We next describe two techniques that boost the optimization performance.

First, although the rate adaptation is only making decisions for a limited window, the search space is still excessive due to the large number of tiles (*e.g.,* up to 72 tiles in a 3-second window for 4×6 tiles). We thus impose two additional constraints: (1) all tiles belonging to the same class should have the same quality level, and (2) the quality level never increases as the class becomes lower. Intuitively, these two constraints reduce the search space by performing rate adaptation on a per-class basis instead of on a per-tile basis, making it feasible for Flare to perform *exhaustive search*. For example, for 5 quality levels and 4 classes (Figure 7), there are only 70 ways to assign quality levels to classes. Moreover, the first constraint helps reduce the quality level changes.

Second, minimizing the stall duration is critical but modeling the exact stall duration is not trivial. Again for performance consideration, we change the stall component from the objective function (Eq. 7) to a series of constraints (one per each scheduled tile) to ensure that none of the tiles incurs stalls. Specifically, let T_0 be the current time and τ_i be the time when the i-th tile in the ordered tile list (derived by the tile scheduler) will appear in a viewport or its associated OOS tile set for the first time. Then we add one constraint for the

i-th tile as follows:

$$\zeta \cdot \mathit{EstBW} \cdot (\tau_i - T_i^{\text{decoding}} - T_0) \geq \sum_{j=1}^{i} \mathit{TileSize}(j) \qquad (8)$$

where *EstBW* is the estimated bandwidth, and $\zeta \in (0, 1]$ is the "damping coefficient" tolerating the bandwidth prediction errors. Eq. 8 dictates that there should be enough time to (1) download all tiles from the first to the i-th tile in the tile list (tiles are transferred sequentially) and to (2) decode the i-th tile. We model the decoding time T_i^{decoding} in §7.2.

After applying the above two optimizations, Flare is able to invoke the rate adaptation at a very fast pace (*e.g.,* for every video frame) on commodity smartphones. The rate adaptation algorithm tests all possible ways of assigning quality levels to classes, and selects the one yielding the highest utility under the aforementioned constraints. If no such an assignment exists, Flare will assign the lowest level to all classes.

Leveraging Buffer Information. The client-side buffer (the Encoded Tile Buffer in Figure 2) contains downloaded tiles that the user is expected to view according to the VP. The buffer has the same length as the VP window, which is $m\delta T$ = 3 seconds in our current implementation. The client will not download any tiles beyond the prediction window due to a lack of VP. A buffered tile that is not consumed by the viewer will be discarded.

Prior studies indicate that considering the client-side buffer status can improve the robustness of DASH rate adaptation [34]. Flare therefore includes a mechanism that leverages the buffer occupancy level to reduce the risk of stalling. We define the buffer occupancy level o as the fraction of predicted Class-0 tiles (*i.e.,* those to appear in any viewport according to the current tile scheduling) that have already been downloaded. Clearly, o may change from time to time as a user moves her head. Having the buffer occupancy level computed, Flare adjusts the damping coefficient ζ in Eq. 8 as $\zeta = o \cdot (Z_{UB} - Z_{LB}) + Z_{LB}$. Intuitively, ζ varies between a lower bound Z_{LB} and an upper bound Z_{UB} depending on o. When the buffer occupancy level is low, ζ decreases. In this way, the tiles' quality levels are lowered, thus reducing the risk of stalling.

6 TILE DECODING AND RENDERING

Let us now consider how to play the tiles after receiving them from the server (the RHS in Figure 2). This is trivial for conventional video playback: each chunk's (only one) tile is sequentially fed into the decoder, whose output frames will then be sequentially rendered at a fixed FPS (frames per second). Flare differs in that a viewport typically contains multiple independently-encoded tiles that are played at the same time. A natural idea is to use multiple decoders, each processing a tile in a *synchronous* manner: all decoders concurrently decode different tiles of frame i in the viewport; when they all finish, the entire frame i is rendered by stitching different decoded portions together; then all decoders move on to frame $i + 1$, and so on. This approach is simple to realize, but suffers from three issues. First, it requires *many* concurrent decoders that a commodity mobile device may not be able to support (§9.4). Second, synchronization also worsens the overall performance, because a decoder finishing early has to wait for other decoders. Third, the key problem occurs when the viewport changes due to head movement. Let us assume that a viewport change occurs at frame f, and that change is large enough so the set of visible tiles will change. For example, a new tile y comes into the viewport. Due to the inter-frame dependency, the decoder cannot directly decode frame f in y. Instead, it must begin with the very first I-frame in y and "fast forward" to f, incurring a potentially long delay.

Our Proposed Approach for Tile Decoding. We realize that the above disadvantages come from the constraint where a decoder decodes only tiles that are *currently* being played[3]. In other words, the playback and decoding are synchronized, and this forces all decoders to be synchronized as well. Our proposed design instead makes decoding and playback asynchronous. It allows decoders to cache the decoded frames of tiles to be played in the future by introducing a *Decoded Frame Buffer* (DFB). Specifically, the *decoding scheduler* (§7.2) dynamically selects a received tile and sends it to an idle decoder. The decoded frames are not necessarily consumed right away; instead they can be stored in the DFB residing in the video memory. When a cached frame is needed during the playback, it is fed into GPU for immediate rendering with negligible delay incurred.

Our proposed design offers several advantages. First, the number of decoders can be greatly reduced; in theory we only need one decoder (if it is fast enough) to decode all tiles. This is because by making decoding and playback asynchronous, we do not need to link each decoder to just one tile. In other words, there is no one-to-one association between visible tiles and decoders. Second, our design ensures smooth playback when visible tiles change, as long as future tiles are properly cached in DFB. Third, we find that asynchronous decoding also dramatically improves the performance compared to synchronous decoding (up to 2.7× higher FPS when tested on SGS 7, using a local 4K video segmented into 2×4 tiles).

A downside of using the DFB is the high video memory usage (§9.6), since the decoded frames are stored in their raw (uncompressed) format. However, Flare only stores a small number of tiles in DFB (as limited by the short prediction window of VP), making our approach feasible for 4K and even 8K videos as to be demonstrated in our evaluation.

[3] Decoders may have their internal buffers, but the buffers are typically very small and not accessible by applications.

Projection and Rendering. We now describe what happens at the end of the streaming pipeline. Flare utilizes a high-precision timer that fires at a fixed frequency (*e.g.,* 30 FPS) to render each frame. When a timer event occurs, the player first checks whether all (sub)frames associated with the tiles in the current viewport are in the DFB. If not, a stall occurs, and one additional timer event will be scheduled when all missing subframes are ready. Otherwise, Flare projects each subframe to draw the corresponding scene in the viewport. This is done by rendering a series of triangle meshes with their vertices being actually projected. After all subframes are rendered, the stitched frame is displayed.

7 OTHER DESIGN ASPECTS OF FLARE

7.1 Client-Server Communication

After a single invocation of tile scheduling and rate adaptation, the client has determined an ordered list of tiles to be fetched as well as each tile's desired quality level. Flare then consolidates them into a *single* request. If this request is different from the most recent one that has been transmitted, Flare encodes the request using delta encoding and transmits it to the server. The bandwidth overhead of the uplink request stream is negligible. Upon the reception of a request, the server immediately starts transmitting the tiles from the beginning of the tile list. The server guarantees that no tile is transmitted more than once.

Due to the short inter-request time, typically the server cannot finish transmitting the entire tile list before a new request arrives. Very importantly, when receiving a new request, the server must *discard* the tile list of the current request that the server might be working on (except for the tile that is being transmitted, if any). This is the *third* mechanism of tolerating VP's inaccuracy (the first two have been described in §4). Intuitively, it leverages the observation that the temporally closer a viewport is from now, the higher accuracy the VP can achieve (§3.2). Therefore, as the server discards out-of-date predictions in the distant future (recall the server transmits the tiles chronologically from near to far), an earlier inaccurate prediction can be potentially "fixed" by a more recent prediction that is more accurate.

7.2 Decoding Scheduler and the Interplay between Decoding and Rate Adaptation

The Decoding Scheduler's job is to select from the tiles waiting at the Encoded Tile Buffer (Figure 2) the most important ones to decode. It *reuses* the tile scheduling results (§4) that contain the predicted tiles sorted chronologically. Specifically, when any decoder becomes idle, the decoding scheduler will send to that decoder a tile with the highest rank in the currently predicted tile list. In other words, the decoding scheduler selects the best estimated tile with the closest playback deadline to appear in the viewport.

Estimating T_i^{decoding} in Eq. 8. Recall that as shown in Figure 2, the received (encoded) tiles are processed in three steps: (1) *waiting*: they wait at the Encoded Tile Buffer when all decoders are busy; (2) *decoding*: some tiles are selected to be decoded; and (3) *rendering*: some decoded tiles stored in DFB are rendered and consumed by the viewer. We find that the *rendering* step takes negligible time. The *decoding* time for a 1-second tile ranges from 0.2 to 0.5 seconds (on SGS8). We model the decoding time of a tile as $\frac{f}{F}D$. F=30 is the number of frames per tile; $f \in (0, F]$ is the first frame in the tile that will appear in the predicted viewport trajectory or any of its OOS tiles; D is the per-tile decoding time (*i.e.,* decoding all F frames in a tile), measured by the player and averaged over the past 20 samples.

We next estimate the *waiting* time. First consider the scenario where a single tile is waiting in the Encoded Tile Buffer. It cannot be decoded until *any* of the decoders becomes idle. This waiting time can be calculated as $w = \min\{w_1, ..., w_p\}$ where p is the number of decoders, and each decoder i will finish decoding its current tile in w_i seconds (a random variable). Now consider a general scenario where q tiles are waiting to be decoded. In the worst case, q is the number of tiles to appear in the next frame's viewport, and we consider the (longest) waiting time for the last of the q tiles to be decoded. This worst-case waiting time is thus qw, whose expectation can be calculated as $\frac{qD}{p+1}$ by assuming each w_i independently follows a uniform distribution on $[0, D]$. In our system, we use a less conservative waiting time of $qw/2$. Then T_i^{decoding} in Eq. 8 is estimated as the sum of the expected waiting time and decoding time:

$$T_i^{\text{decoding}} = D\left[\frac{f}{F} + \frac{q}{2(p+1)}\right] \tag{9}$$

In Eq. 9, F and p are constants; q can also be approximated as a constant (*e.g.,* we use 6 for 4×6 segmentation); f is derived by the tile scheduler; D can be estimated from measurements at runtime as described before.

8 IMPLEMENTATION

We have fully implemented Flare on commodity Android smartphones and Linux OS. The player is written in Java using Android SDK (for decoding, rendering, projection, tracking head movement, and UI) and C++ using Android NDK (for networking, tile scheduling, rate adaptation, VP, and the Encoded Tile Buffer). Tile decoding is realized through the low-level Android `MediaCodec` API [3]. We use OpenGL ES for GPU-assisted viewport rendering. The DFB is realized through OpenGL ES textures stored in video memory. We have successfully tested Flare on three devices: SGS7, SGS8, and Samsung Galaxy J7 (SGJ7), all running stock Android 7.0 OS. The phones do *not* need to be rooted. The server is implemented in C/C++ on standard Linux platform (Ubuntu

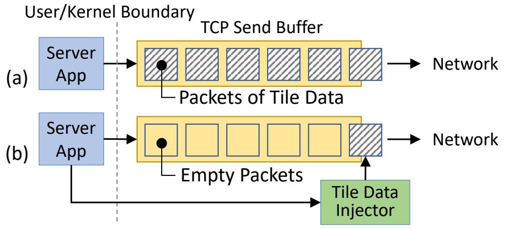

Figure 8: **Two server-side transmission schemes. Flare uses (b).**

14.04 and 16.04). Our implementation consists of 14,200 lines of code (LoC) whose breakdown is as follows: for the client-side player, 4,800 LoC in Java and 5,700 LoC in C++; for the server, 2,900 LoC in C++ (user space server logic) and 800 LoC in C (a kernel module to be described below).

Impact of Server-Side Sender TCP Buffer. The client-server communication channel (§7.1) is realized using a persistent TCP connection. Interestingly, we find that on the server side, its TCP send buffer (`sendbuf`) may severely impact the performance of Flare. On wireless networks, the TCP `sendbuf` may inflate quite a bit due to bufferbloat [30, 35]. Figure 8(a) plots the scheme where Flare runs on unmodified Linux TCP/IP stack. As shown, the TCP `sendbuf` is filled with data packets containing tile contents. At this moment, assume the server receives a new request containing an up-to-date tile list. If the tiles that are queued in `sendbuf` do not appear in the new request, then they should be removed from the `sendbuf`. However, such a removal operation is not supported by TCP, whose `sendbuf` is inherently FIFO. As a result, the unneeded tiles in `sendbuf` cause head-of-line (HoL) blocking and bandwidth waste [45].

Ideally, the above problem should be fixed by using a custom transport protocol. However, given TCP's absolute dominance in today's transport protocols, we design and implement a solution that eliminates the HoL blocking while working with off-the-shelf TCP. As shown in Figure 8(b), in our solution, the server application generates empty packets as "placeholders", and controls a kernel module that injects the actual data only when packets are leaving `sendbuf`. In this way, the aforementioned HoL is eliminated because inside `sendbuf` there are only placeholders and the actual data injection is postponed to the latest possible stage. The kernel module was implemented using the Linux Packet Filtering framework [6]. It properly determines the number of empty packets (bytes) to generate, and also takes care of TCP retransmission, packet reordering, and checksum rewriting. In the above scheme, some bytes may be wasted (*i.e.,* not filled with the actual tile data). In practice, such wasted bytes account for a negligible fraction (<0.5%) of all transmitted bytes.

9 EVALUATION

We extensively evaluate the performance of Flare and compare it with other algorithms. The basic approach is to replay head movement traces from real users for these algorithms.

9.1 Experimental Setup

Video and User Selection. Our user study consists of a large number of users (130) and videos (10). From them, we pick "representative" ones in terms of their VP difficulties. Specifically, we select 6 videos with different VP accuracies including 2 "easy" videos: car racing (3'01") and roller coaster (1'57"), 2 "medium" videos: wild animal (2'49") and skydive (3'53"), and 2 "difficult" videos: concert (4'25") and sea world (2'14"). All videos are in 4K resolution, and are encoded into 5 levels (0 to 4, with 0 being the lowest quality) using different Constant Rate Factor (CRF) values from 23 to 42. The encoded bitrate ratio between two consecutive quality levels is roughly 1:1.5, following prior recommendations [8, 12]. Regarding the users, we uniformly sample 11 users based on their average VP accuracy across all videos. The sampled users' VP accuracy ranges from 63% to 89% (using 1-second prediction window, linear regression).

Network Conditions. For WiFi experiments, our client device (SGS 8) and server (3.6GHz Quad-core CPU, 16GB memory) are connected to a commodity 802.11 AP at 2.4GHz. The end-to-end peak throughput is ∼90Mbps and ping latency is ∼1ms. We then use the Linux `tc` tool to emulate challenging network conditions as follows. We replay 10 bandwidth traces collected from real 4G/LTE networks. Within the 10 traces, 5 are obtained from a public 4G/LTE trace captured during mobility [15]; the other 5 traces are captured by ourselves at various public locations including restaurant, hotel, coffee shop, airport, and shopping mall. Since the bandwidth in some traces is quite high, we linearly scale each trace to make the average bandwidth around 9.6Mbps (standard deviation 4.5Mbps across all 1-second slots). `tc` also inflates the end-to-end latency to 50ms. Similar replay-based network emulation approaches are used in numerous prior studies [31, 44, 59]. For LTE experiments, we directly run Flare on commercial LTE networks at 9 locations in 6 U.S. states with diverse LTE signal strength to be detailed in §9.3.

Algorithms to Compare. We implement the following five algorithms: (1) our Flare algorithm with 4×6, 4×4, and 2×4 tile segmentations, (2) Festive [36], a throughput-based rate adaptation algorithm, (3) BBA [34], a buffer-based rate adaptation algorithm, (4) Full [29], an algorithm that fetches tiles in the viewport with the highest possible quality and all other tiles with the lowest possible quality, and (5) H2 [47], an extension of Full that uses the residual network capacity to increase the quality of tiles adjacent to viewport tiles. Within the above, (1), (4), and (5) are viewport-adaptive 360°

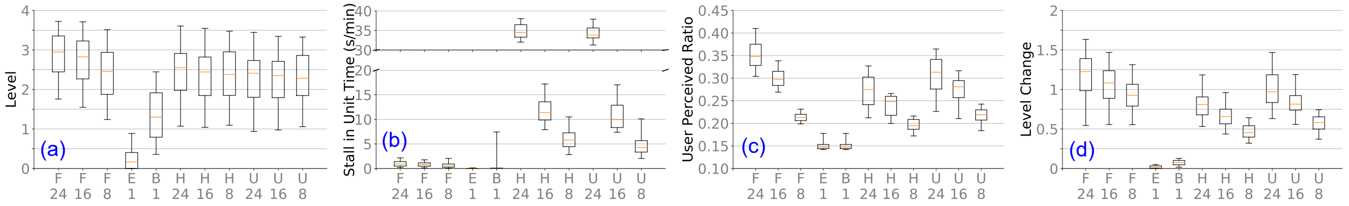

Figure 9: Comparing 5 streaming schemes over WiFi with limited bandwidth. *Schemes*: F=Flare, E=Festive, B=BBA, H=H2, U=Full. *Segmentation*: 24=4×6 tiles, 16=4×4, 8=2×4, 1=1×1 tile. *Metrics*: (a) perceived quality level, (b) stall per minute, (c) user perceived ratio, (d) quality level changes.

streaming algorithms; (2) and (3) are algorithms for conventional DASH videos. Note that all viewport-adaptive schemes (Flare, H2, and Full) use the same buffer size (3 seconds' worth of tiles). Both non-viewport-adaptive schemes (Festive and BBA) employ a longer buffer size of 40 seconds. Their non-viewport-adaptive nature allows the buffer size to be longer. We use the four evaluation metrics defined in §5, plus the total downloaded bytes to assess the above algorithms. The quality switch is computed as the sum of Eq. 5 and Eq. 6.

Parameter Selection. For Flare, we empirically determine the following parameters: 4 parallel H.264 decoders, ξ=1 (Eq. 1), α=0.5 (Eq. 2), w_i=1 (Eq. 7), Z_{UB}=0.9, Z_{LB}=0.3 (§5), F_x=100°, F_y=90°, and four classes constructed as shown in Figure 7. We study their impacts in §9.9. For other algorithms, we use the recommended parameters in their original papers.

9.2 WiFi Results

Perceived Quality Level. Figure 9(a) plots for each streaming scheme the average quality level of consumed tiles (Eq. 4) across all playbacks. Recall in §9.1 that each bar consists of 660 video playbacks (6 videos × 11 users × 10 bandwidth traces, about 34 hours' playback). Flare's 4×6, 4×4, and 2×4 tile schemes yield median quality levels of 2.95, 2.83, and 2.46, respectively. Festive and BBA only achieve 0.164 and 1.29, respectively, due to their viewport-agnostic approaches. Overall, Flare's median quality level improvement ranges from 1.90× to 18.0×. H2 and Full are also viewport-adaptive, but they yield slightly lower quality level compared to Flare, because they download all tiles (more than necessary).

Stall. Figure 9(b) shows distributions of stall duration, measured in *seconds per playback minute*, across all playbacks. Festive and BBA yield the lowest stall, while Flare's stall is a bit longer: for 4×6, 4×4, and 2×4, the median stall across 660 playbacks are 0.96, 0.80, and 0.55 s/min, respectively. This is again due to the viewport-adaptive paradigm employed by Flare: fetching a subset of tiles naturally increases the probability of stalls. However, due to Flare's various design decisions at both algorithm and system levels, we manage to reduce the stall to a very low level (median value less than 1s/min). Meanwhile, we do observe that in some cases BBA incurs high stall due to scarce bandwidth and BBA's aggressive quality selection strategy (compared to Festive). Also, as expected, a large tile size for Flare can better tolerate inaccurate viewport prediction and thus reduce the stall (§4).

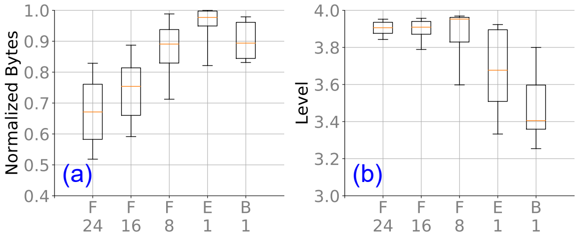

Figure 10: Comparing 3 schemes over WiFi without bandwidth throttling: (a) normalized downloaded bytes, (b) perceived quality level.

It is worth mentioning that the stall of Flare can be further reduced. For example, an effective approach would be to always download a 1×1 chunk containing the entire panoramic view at the *lowest* resolution or quality level to guarantee there is always some content to play even when the VP is inaccurate. This approach can be easily integrated into Flare.

Figure 9(b) also illustrates that both H2 and Full suffer from unacceptably high stall duration. This is because (our best-effort implementations of) H2 and Full do not have Flare's salient features such as multi-timestamp VP (§4), fast-paced tile scheduling (§4), and decoding schedulers (§7.2). As a result, despite their utilization of the tile-based streaming concept and VP, their overall system performance is poor.

User Perceived Ratio. Figure 9(c) shows the ratio between the consumed and the overall video bitrate, across all playbacks. The results cross-validate those in Figure 9(a): the user perceived ratios for Flare significantly outperform those of Festive and BBA.

Quality Level Changes. Figure 9(d) plots the quality level changes (the sum of Eq. 5 and Eq. 6) across all playbacks. As shown, Flare has higher quality level changes compared to other schemes. However, based on our viewing experiences, its incurred QoE degradation is very limited because most quality level changes (both inter- and intra-chunk) occur at the periphery of the viewport (recall in §5 that Flare enforces that all Class-0 tiles have the same quality level); also the changes are oftentimes barely noticeable due to the tiles' small overlap with the viewport. Moreover, the quality changes can further be reduced by tuning w_i in Eq. 7, as to be shown in §9.9. Note that a recent study [54] investigates the perceptual effect

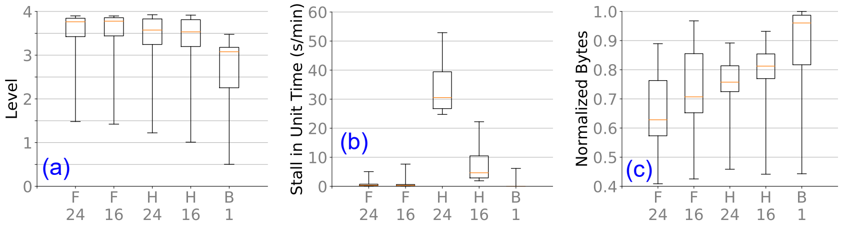

Figure 11: **Comparing 3 streaming schemes over commercial LTE networks: (a) perceived quality level, (b) stall, and (c) normalized downloaded bytes.**

		F 4×4	H 4×6	H 4×4	B 1×1
Office	Level	1.01	0.96	0.95	0.83
	Stall	0.79	89.7	9.62	0.03
	DL	1.12	1.14	1.20	1.38
Hotel	Level	0.86	0.60	0.60	0.17
	Stall	1.16	5.34	2.34	1.29
	DL	1.01	1.15	1.16	1.08
Conf Center	Level	1.02	0.88	0.93	0.54
	Stall	1.03	30.2	11.5	1.74
	DL	1.13	1.11	1.24	1.03

Table 2: **LTE results at 3 locations. Comparing 5 schemes on 3 metrics.** Flare **4×6 is the baseline (1.00).**

on viewers when adjacent tiles have different video encoding quality levels via a psychophysical study with 50 participants. They found that mixed-resolution tiles are acceptable for most users for videos with low and medium motion.

Bandwidth Savings. We now quantify the bandwidth reduction brought by Flare. We repeat the experiments in Figure 9 under an ideal network condition where the `tc` bandwidth throttling is removed. The two subplots in Figure 10 show the distributions of per-video download sizes (normalized for each video) and perceived quality levels. Each bar consists of 66 playbacks (6 videos × 11 users, about 3.4 hours' playback). As shown in Figure 10(a), compared to Festive and BBA, Flare reduces the bandwidth consumption (the median across all 66 runs) by up to 31.1%, depending on the segmentation configuration. Note that the bandwidth savings are conservative estimations because, as shown in Figure 10(b), even when the bandwidth is high, Festive and BBA's quality level selections are lower than Flare.

9.3 Commercial LTE Results

We conduct extensive experiments on commercial LTE networks of a large U.S. cellular carrier. We select 9 locations in 6 states. They include hotels, apartments, residential houses, conference centers, and office buildings. In each location, we replay 4 diverse users' head movement traces for 3 videos (one "easy" video: roller coaster; one "medium": skydive; and one "difficult": sea world), using five schemes (Flare with 4×6 and 4×4, H2 with 4×6 and 4×4, BBA). The experiments were repeated multiple times at different time of day. They span ∼100 hours, consuming over 600 GB cellular data.

The three subplots in Figure 11 show the distributions of average bitrate level, stall, and downloaded bytes (normalized for each video) across all locations, users, and videos. Overall, as shown in Figure 11(a), compared to BBA, Flare increases the average bitrate level by 22.7% and 22.3% for 4×4 and 4×6 tiles, respectively. The improvement is smaller than those in Figure 9 because of the high LTE bandwidth in many (6 out of 9) places. Nevertheless, as shown in Figure 11(c), with such quality improvement, Flare also reduces the downloaded bytes by 26.4% and 34.6% for 4×4 and 4×6 tiles, compared to BBA. Another encouraging result is reported in Figure 11(b): Flare achieves qualitatively similar stall compared to BBA in particular when the bandwidth is constrained. The results of H2 are largely inlined with the WiFi results (§9.2) as exhibited by high stall duration.

Table 2 presents case studies for three locations. All numbers are normalized using Flare 4×6 as the baseline (1.00). For *office*, the network connectivity is good so Flare can help reduce the bandwidth consumption by 28% while increasing the quality level by 20%. For *hotel* where the LTE bandwidth is lower, Flare effectively improves the consumed quality level by 4.9× and reduces the stall time by 22% compared to BBA. The *conference center* has even worse network conditions due to poor signal strengths (around -116dBm) and dense users. In this scenario, BBA cannot even support the lowest quality (level 0), while the average quality for Flare almost doubles with 43% reduction of the stall time.

9.4 Decoder Parallelism

Figure 12 studies the impact of the number of decoders. Each bar consists of 22 playbacks (2 videos: roller coaster and wild animal; all 11 users selected in §9.2) for 4×6 tiles on SGS8. The number of hardware decoders has a big impact on the stall time. On one hand, we indeed need to leverage parallel decoders on commodity smartphones, as indicated by the high stall time of using only one decoder. On the other hand, using too many decoders also negatively affects the performance: the complex video decoding pipeline involves multiple hardware subsystems on a smartphone SoC (system-on-a-chip) such as CPU, GPU, and DSP [17]; any of them may potentially become the resource bottleneck when an excessive number of decoding threads are running. We select 4 decoders for SGS8 based on Figure 12. The selection process can be automated through one-time benchmarking on a new device.

9.5 Server-Side Tile Transmission Scheme

Recall in §8 that Flare introduces an OS kernel mechanism that "bypasses" TCP `sendbuf`. How important is this feature? In Figure 13, we consider three server-side tile transmission schemes. "Usr+Knl Update" is Flare's mechanism as illustrated in Figure 8(b). "User Update" is the transmission mode without the kernel mechanism as depicted in Figure 8(a);

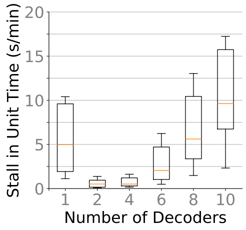

Figure 12: Impact of # of parallel decoders.

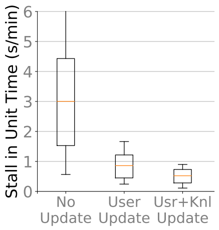

Figure 13: Server transmission schemes.

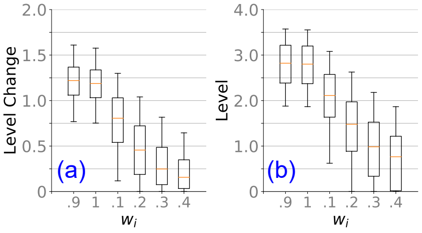

Figure 14: Impact of w_i on (a) level change and (b) level quality (simulation).

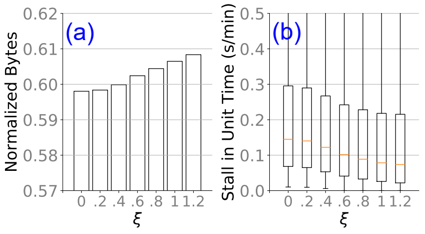

Figure 15: Impact of ξ on (a) downloaded bytes and (b) stall time (simulation).

in this mode, an out-of-date tile can only be removed from server's app-layer transmission buffer in the *user space*. "No Update" is a simple scheme where once a tile is pumped into the app-layer buffer, it is never removed. Each bar in Figure 13 consists of 60 playbacks (2 videos × 3 users × all 10 bandwidth traces selected in §9.2). As shown, updating tile content in the buffer leads to significant reduction of the stall time, due to the elimination of head-of-line blocking incurred by out-of-date tiles. Also, such update needs to be performed in both the user and kernel buffers: our designed kernel mechanism effectively reduces the median stall duration by 39% compared to the "User Update" scheme.

9.6 Video Memory Consumed by DFB

Flare needs video memory to store decoded frames in the Decoded Frame Buffer (DFB, §6). Our current implementation manages the DFB as follows. The DFB maintains k slots each can store a decoded frame. The slots are allocated in a "lazy" manner. At the beginning of a video playback, k is initially set to 10. When any decoder produces a frame of a tile, the frame is saved into an empty slot. If there is no empty slot, k is increased by 10 (*i.e.,* additional 10 slots are allocated). When a frame is consumed or discarded, its slot is marked as empty so the storage can be reused, but k is not decreased.

To quantify the video memory overhead of DFB, we record k *after* each of the 660 playbacks in §9.2. The recorded value thus corresponds to the maximum number of allocated slots. We can then compute the maximum video memory usage of this playback, denoted as $V_{\max}$, as the product of k, the tile's width × height, and the color-depth of each pixel. For 4×6 segmentation, the 25th, 50th, 75th, and 90th percentile of $V_{\max}$ are 596MB, 636MB, 721MB, and 852MB, respectively. Overall, the video memory usage is indeed non-trivial, but it well fits the available video memory, which is typically shared with the main memory, of today's COTS smartphones.

9.7 Energy and Thermal Overhead

We conduct the following experiments to measure the energy consumption of Flare (4×6 tiles) and compare it with BBA. We fully charge an SGS8 phone and play a video (Roller Coaster in 4K) continuously for 30 minutes, and then read the battery percentage level. Over LTE networks with good signal strength, after 30-minute continuous playback, the battery level drops to 86% and 92% for Flare and BBA, respectively. Flare consumes more energy largely because of its higher CPU usage (average utilization of 58% compared to 18% for BBA). The GPU usage is similar for Flare and BBA (11% vs. 8%). Over WiFi, the battery level drops to 88% and 94% for Flare and BBA after 30-minute playback. Regarding the thermal overhead, Flare yields slightly higher CPU temperature compared to BBA (48°C vs. 44°C over LTE and 46°C vs. 40°C over WiFi). Overall, we believe the above overhead is non-trivial but acceptable. They can potentially be reduced by better system-level engineering or edge-cloud offloading.

9.8 Other Results from the Real System

Other Devices. So far all our experiments run on SGS8. We also test Flare on two older devices: Samsung Galaxy S7 (SGS7) and Samsung Galaxy J7 (SGJ7). SGS7 offers comparable performance to SGS8. Unlike SGS8/SGS7, SGJ7 is a medium-end smartphone released almost 3 years ago. On SGJ7, we do observe higher stall time (up to 1.7× higher than SGS8 under the same network condition and user head movement trajectory) due to the longer decoding time and less frequent invocations of tile scheduling and rate adaptation, caused by its limited computation power. We believe Flare will offer better performance on future mobile devices.

8K Videos. Flare is a general framework that can stream videos in any resolution and encoding scheme. To test its performance on 8K videos, we use the *360° Angel Falls* video [14] that has been viewed 2.3M times on YouTube as in March 2018. The video's original resolutions are 7680×3840 and 3840×2160 for 8K and 4K respectively. We segment them into 4×6 tiles at a high quality (CRF=23). Both 4K and 8K can be played on SGS8 (we test them over 50Mbps WiFi and a user's head movement trajectory with the VP accuracy being around 80%). For 4K, the stall is around 0.2s/min, while for 8K, the stall is higher at around 2.2s/min due to the high decoding overhead. This is confirmed by comparing the decoding penalty averaged over T_i^{decoding} defined in Eq. 9: the penalties for 4K and 8K playback are 1:2.34. Note that Flare's current implementation uses H.264 encoding; it may achieve better performance through newer encoding schemes such as H.265/HEVC with better support for parallel decoding [46].

9.9 Trace-driven Simulations

To further scale up our evaluations, we also develop a simulator for Flare and other streaming algorithms. The simulator takes as input real bandwidth traces (we select 20 traces using the same method described in §9.1) and users' head movement traces (we use all 130 users in our user study), and computes the key performance metrics defined in §5 for all 10 videos. We repeat the experiments in Figure 9 and 10 on our simulator and observe qualitatively similar results.

Parameter Selection. Leveraging our simulator, we systematically explore the impact of various parameters listed in §9.1. Here we exemplify two key parameters: w_i (controlling the importance of quality change in Eq. 7) and ξ (controlling the aggressiveness of fetching OOS tiles in Eq. 1). As shown in Figure 14(a), as we increase w_i, the tiles' quality levels become homogeneous; meanwhile, as shown in Figure 14(b), the overall playback quality decreases. This is because it is difficult to increase the playback quality for an individual tile as doing so would require increasing other tiles' qualities as well. Figure 15 studies the impact of ξ. As shown, increasing ξ causes slight inflation of bandwidth consumption due to the additionally downloaded OOS tiles; on the positive side, doing so helps reduce the stall duration.

10 RELATED WORK

There are several proposals of viewport-adaptive streaming for 360° videos. Bao *et al.* [23] propose a partial content transmission scheme using head movement prediction; Qian *et al.* [48] propose to explicitly use a tile-based approach; 360ProbeDASH [57] is another QoE-driven optimization framework for tile-based streaming. There are other recent proposals such as [28, 29, 33, 42, 47, 61]. There are also proposals on various projection and content representation schemes for 360° videos. For example, POI360 [58] is an interactive live 360° video streaming system that dynamically adjusts the compression strategy to maintain good perceived quality. Li *et al.* [40] propose a tile-based encoding solution for omnidirectional videos by creating approximately equal-area titles after projection. Other studies along this direction include [26, 54, 60], to name a few. Compared to the above studies, Flare consists of a holistic framework for the end-to-end 360° video streaming pipeline. It introduces new streaming algorithms, and brings many system and network-level optimizations that have not been investigated previously.

Recently, He *et al.* developed Rubiks, another practical 360° video streaming system for smartphones [32]. It also employs the tile-based streaming paradigm and viewport prediction. Flare differs from Rubiks in several aspects, to list a few below. First, Rubiks requires special encoding that temporally splits a tile, while Flare can work with traditional encoding schemes such as unmodified H.264 and H.265, which produce encoded tiles with smaller sizes compared to tiles in Rubiks. Second, unlike Rubiks that conducts single-point VP, Flare predicts the viewport *trajectory*, and allows a previously inaccurate prediction to be updated by a more recent prediction. Third, Flare's realization of the rate adaptation is different from Rubiks, offering additional features such as multi-class tiles and buffer-aware rate adaptation. Another recent tile-based 360° video streaming system is BAS-360 [56]. Compared to Flare, BAS-360 misses several key components such as VP and efficient decoding. In addition, neither work was evaluated over real cellular networks as we did for Flare.

360° videos play an important role in the VR ecosystem. Tremendous efforts have been made towards mobile VR. Recently developed VR systems include FlashBack [24], MoVR [20], Furion [39], and LTE-VR [53], to name a few. They are orthogonal to and can benefit from Flare. For example, our VP algorithm can improve the prefetching performance for both FlashBack and Furion.

11 LIMITATIONS AND CONCLUSION

Flare is just our first step towards intelligent 360° video streaming. In its current design, the system still has a few limitations that we plan to address in our future work.

A key challenge of viewport-adaptive streaming is VP. Despite a series of optimizations, Flare may still suffer from higher stalls than non-viewport-adaptive approaches due to imperfect VP. There are several future research directions: (1) using cloud/edge to support more advanced VP algorithms, (2) exploring crowd-sourced head movement data [42] and video content [27] for long-term VP, and (3) using other sensors such as gaze-tracking [38, 41] to improve VP.

Another concern of tile-based streaming is its incurred higher quality changes. Our rate adaptation algorithm has built-in features for mitigating them and offering knobs to tune them (Figure 14). However, the impact of such quality changes on QoE and QoE metrics for 360° videos in general are still under-explored, compared to the well-studied QoE metrics for regular videos [22].

Fetching the entire panoramic content has other advantages such as being able to quickly rewind and look at the scene from another view. Realizing these features using viewport-adaptive approaches involves extra complexities.

To conclude, Flare demonstrates that it is feasible to realize the tile-based concept for streaming 4K and even 8K 360° videos on commodity mobile devices without additional infrastructures or special video encoding support. We intend to make our head movement dataset publicly available to facilitate intra-disciplinary research in this space.

ACKNOWLEDGEMENTS

We would like to thank the voluntary users who participated in our user study, and Justus Schriedel for managing the study. We also thank the anonymous MobiCom reviewers and the shepherd for their valuable comments. Feng Qian's research was supported in part by NSF Award #1566331, #1629347, an AT&T VURI Award, and a Google Faculty Award.

REFERENCES

[1] 360 Google Spotlight Story: Help . https://www.youtube.com/watch?v=G-XZhKqQAHU.

[2] 360° Great Hammerhead Shark Encounter . https://www.youtube.com/watch?v=rG4jSz_2HDY.

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

[4] Elephants on the Brink . https://www.youtube.com/watch?v=2bpICIClAIg.

[5] GT-R Drives First EVER 360 VR lap . https://www.youtube.com/watch?v=LD4XfM2TZ2k.

[6] Linux Packet Filtering framework . https://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html.

[7] Mega Coaster: Get Ready for the Drop . https://www.youtube.com/watch?v=-xNN-bJQ4vI.

[8] Per-Title Encode Optimization . https://medium.com/netflix-techblog/per-title-encode-optimization-7e99442b62a2.

[9] Pony Stable Playhouse for the Currys . https://www.youtube.com/watch?v=MWg1kjMmr3k.

[10] Under the hood: Building 360 video . https://code.facebook.com/posts/1638767863078802.

[11] Visit Hamilton Island in 360° Virtual Reality with Qantas . https://www.youtube.com/watch?v=lJype_TafRk.

[12] What Is Per-Title Encoding? . https://bitmovin.com/per-title-encoding/.

[13] YouTube live in 360 degrees encoder settings . https://support.google.com/youtube/answer/6396222.

[14] 360° , Angel Falls, Venezuela . https://www.youtube.com/watch?v=L_tqK4eqelA.

[15] 4G/LTE Bandwidth Logs. http://users.ugent.be/~jvdrhoof/dataset-4g/.

[16] Feel wimbledon with andy murray. https://www.youtube.com/watch?v=Krl6U15OERo.

[17] Qualcomm Snapdragon 835. https://www.qualcomm.com/products/snapdragon/processors/835.

[18] Skydive in 360° virtual reality via gopro. https://www.youtube.com/watch?v=S5XXsRuMPIU.

[19] Tomorrowland 2014 — 360 degrees of madness. https://www.youtube.com/watch?v=j81DDY4nvos.

[20] O. Abari, D. Bharadia, A. Duffield, and D. Katabi. Enabling High-Quality Untethered Virtual Reality. In *Proceedings of NSDI 2017*, pages 531–544. USENIX Association, 2017.

[21] S. Afzal, J. Chen, and K. Ramakrishnan. Characterization of 360-Degree Videos. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pages 1–6. ACM, 2017.

[22] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *Proceedings of SIGCOMM 2013*, pages 339–350. ACM, 2013.

[23] Y. Bao, H. Wu, T. Zhang, A. A. Ramli, and X. Liu. Shooting a Moving Target: Motion-Prediction-Based Transmission for 360-Degree Videos. In *Proceedings of Big Data 2016*, pages 1161–1170. IEEE, 2016.

[24] K. Boos, D. Chu, and E. Cuervo. FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization. In *Proceedings of MobiSys 2016*, pages 291–304. ACM, 2016.

[25] X. Corbillon, F. De Simone, and G. Simon. 360-Degree Video Head Movement Dataset. In *Proceedings of MMSys 2017*. ACM, 2017.

[26] X. Corbillon, G. Simon, A. Devlic, and J. Chakareski. Viewport-Adaptive Navigable 360-Degree Video Delivery. In *Proceedings of ICC 2017*. IEEE, 2017.

[27] C.-L. Fan, J. Lee, W.-C. Lo, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu. Fixation Prediction for 360 Video Streaming in Head-Mounted Virtual Reality. In *Proceedings of the Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 67–72. ACM, 2017.

[28] 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.

[29] M. Graf, C. Timmerer, and C. Mueller. Towards bandwidth efficient adaptive streaming of omnidirectional video over HTTP: Design, implementation, and evaluation. In *Proceedings of MMSys 2017*, pages 261–271. ACM, 2017.

[30] Y. Guo, F. Qian, Q. A. Chen, Z. M. Mao, and S. Sen. Understanding on-device bufferbloat for cellular upload. In *Proceedings of IMC 2016* , pages 303–317. ACM, 2016.

[31] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath. In *Proceedings of CoNEXT 2016*, pages 129–143. ACM, 2016.

[32] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han. Rubiks: Practical 360-Degree Streaming for Smartphones. In *Proceedings of MobiSys 2018*. ACM, 2018.

[33] M. Hosseini and V. Swaminathan. Adaptive 360 VR video streaming: Divide and conquer. In *Multimedia (ISM), 2016 IEEE International Symposium on*, pages 107–110. IEEE, 2016.

[34] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of SIGCOMM 2014*, pages 187–198. ACM, 2014.

[35] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G networks . In *Proceedings of IMC 2012*, pages 329–342. ACM, 2012.

[36] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Video Streaming With Festive. In *Proceedings of CoNEXT 2012*, pages 97–108. ACM, 2012.

[37] N. Jiang, V. Swaminathan, and S. Wei. Power Evaluation of 360 VR Video Streaming on Head Mounted Display Devices. In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 55–60. ACM, 2017.

[38] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving User Perceived Page Load Times Using Gaze . In *Proceedings of NSDI 2017*, pages 545–559. USENIX Association, 2017.

[39] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proceedings of MobiCom 2017*, pages 409–421. ACM, 2017.

[40] J. Li, Z. Wen, S. Li, Y. Zhao, B. Guo, and J. Wen. Novel tile segmentation scheme for omnidirectional video. In *Proceedings of ICIP 2016*, pages 370–374. IEEE, 2016.

[41] T. Li, Q. Liu, and X. Zhou. Ultra-Low Power Gaze Tracking for Virtual Reality . In *Proceedings of SenSys 2017*. ACM, 2017.

[42] X. Liu, Q. Xiao, V. Gopalakrishnan, B. Han, F. Qian, and M. Varvello. 360 Innovations for Panoramic Video Streaming . In *Proceedings of HotNets 2017*. ACM, 2017.

[43] W.-C. Lo, C.-L. Fan, J. Lee, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu. 360 Video Viewing Dataset in Head-Mounted Virtual Reality. In *Proceedings of MMSys 2017*, pages 211–216. ACM, 2017.

[44] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve . In *Proceedings of SIGCOMM 2017*, pages 197–210. ACM, 2017.

[45] X. Mi, F. Qian, and X. Wang. SMig: Stream Migration Extension For HTTP/2. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 121–128. ACM, 2016.

[46] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth, and M. Zhou. An overview of tiles in HEVC. *IEEE Journal of selected topics in signal processing*, 7(6):969–977, 2013.

[47] S. Petrangeli, V. Swaminathan, M. Hosseini, and F. De Turck. An HTTP/2-based adaptive streaming framework for 360° virtual reality videos. In *Proceedings of MM 2017*, pages 1–9. ACM, 2017.

[48] F. Qian, B. Han, L. Ji, and V. Gopalakrishnan. Optimizing 360 video delivery over cellular networks. In *Proceedings of the Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 1–6. ACM, 2016.

[49] R. Real and J. M. Vargas. The probabilistic basis of Jaccard's index of similarity. *Systematic biology*, 45(3):380–385, 1996.

[50] C. Saunders, A. Gammerman, and V. Vovk. Ridge Regression Learning Algorithm in Dual Variables . In *Proceedings of ICML 1998*, pages 515–521, 1998.

[51] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

[52] 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 *Proceedings of ISM 2016*, pages 583–586. IEEE, 2016.

[53] Z. Tan, Y. Li, Q. Li, Z. Zhang, Z. Li, and S. Lu. Enabling Mobile VR in LTE Networks: How Close Are We? In *Proceedings of SIGMETRICS 2018*. ACM, 2018.

[54] H. Wang, V.-T. Nguyen, W. T. Ooi, and M. C. Chan. Mixing tile resolutions in tiled video: A perceptual quality assessment. In *Proceedings of the Workshop on Network and Operating System Support on Digital Audio and Video*, page 25. ACM, 2014.

[55] C. Wu, Z. Tan, Z. Wang, and S. Yang. A Dataset for Exploring User Behaviors in VR Spherical Video Streaming. In *Proceedings of MMSys 2017*, pages 193–198. ACM, 2017.

[56] M. Xiao, C. Zhou, V. Swaminathan, Y. Liu, and S. Chen. BAS-360: Exploring Spatial and Temporal Adaptability in 360-degree Videos over HTTP/2. In *INFOCOM 2018-IEEE Conference on Computer Communications, IEEE*. IEEE, 2018.

[57] L. Xie, Z. Xu, Y. Ban, X. Zhang, and Z. Guo. 360ProbDASH: Improving QoE of 360 Video Streaming Using Tile-based HTTP Adaptive Streaming. In *Proceedings of MM 2017*, pages 315–323. ACM, 2017.

[58] X. Xie and X. Zhang. POI360: Panoramic Mobile Video Telephony over LTE Cellular Networks. In *Proceedings of CoNEXT 2017*, pages 336–349. ACM, 2017.

[59] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of SIGCOMM 2015*, pages 325–338. ACM, 2015.

[60] M. Yu, H. Lakshman, and B. Girod. A framework to evaluate omnidirectional video coding schemes. In *Proceedings of the Symposium on Mixed and Augmented Reality (ISMAR) 2015*, pages 31–36. IEEE, 2015.

[61] A. Zare, A. Aminlou, M. M. Hannuksela, and M. Gabbouj. HEVC-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proceedings of MM 2016*, pages 601–605. ACM, 2016.

[62] C. Zhou, Z. Li, and Y. Liu. A Measurement Study of Oculus 360 Degree Video Streaming. In *Proceedings of MMSys 2017*. ACM, 2017.