ClusTile: Toward Minimizing Bandwidth in 360-degree Video Streaming

Chao Zhou
SUNY Binghamton
czhou5@binghamton.edu

Mengbai Xiao
George Mason University
mxiao3@gmu.edu

Yao Liu
SUNY Binghamton
yaoliu@binghamton.edu

***Abstract*—360-degree video has the potential to transform the video streaming experience by providing a more-immersive environment for users to interact with than standard streaming video. This experience is hampered, however, by high bandwidth requirements resulting from the extra information associated with the 360-degree frames. Because users cannot see this full 360-degree view, but the full view is transmitted in the majority of 360-degree streaming systems, there is much potential to reduce wasted bandwidth in this domain.**

We propose ClusTile, a tiling approach formulated to select a set of tiles that allows minimal bandwidth needed to be used when streaming 360-degree video over an expected set of views. These tiles are selected by solving a set of integer linear programs (ILPs) independently on clusters of collected user views. The clustering approach reduces computation requirements of the ILPs to practical levels. Tilings computed from ClusTile can save up to 76% bandwidth compared to standard 360-degree streaming and up to 52% bandwidth compared to best-performing fixed tiling schemes.

I. Introduction

360-degree video delivers an immersive view to the viewer, allowing exploration of all view orientations emanating from the camera position. Increasingly sophisticated and inexpensive hardware (e.g., Samsung's GearVR) enable the 360-degree video experience. Although it is possible to view 360-degree videos on standard video players, head mounted displays with motion tracking render these 360-degree videos truly immersive. The combination of new hardware and novelty of the 360 degree video mean that it has the potential to see widespread adoption in the coming years. Despite this potential, the software and algorithms needed to stream 360-degree video are still relatively immature compared to standard video streaming. These software deficiencies could prevent mainstream acceptance as bandwidth requirements for video streaming might be too high for many users.

To illustrate the problem, 360-degree video streaming on a VR device likely requires 4K resolution equirectangular frames, a common 360-degree encoding, to achieve an adequate viewing experience [10]. This high resolution is necessary due to the combination factors. Close device proximity to the user's eyes means that high resolution is needed at viewing time. Stereo viewing halves resolution from the delivered 360-degree frame, and, finally, only a small portion of each 360-degree frame is displayed on the screen. For example, a 100- by 100-degree view centered in an equirectangular frame covers only about 14.3% of the frame's pixels.

Netflix's minimum recommended 4K video connection speed is 25 Mbps [6], but average broadband speeds in the USA are only 15.3 Mbps [1]. This bandwidth discrepancy could pose problems for widespread 360-degree video adoption without advances over the current status quo.

At least three core technical challenges in 360-degree video streaming are related to this bandwidth problem: i) The 360 degree view cannot be directly encoded in a rectangular frames, as is the case for standard videos. ii) Users can only see a small field of view within the full 360-degree frame, but these full frames are typically delivered during streaming, wasting the unviewed portion of the bandwidth. iii) During streaming the client does not know the user's future orientation, making selection of an area of the 360-degree view difficult. Note that (iii) depends on (ii) in that predicted user orientations are only useful if the 360-degree video streaming infrastructure can support downloading of sub-areas that reflect the prediction.

Of these problems, this work focuses on item (ii). Specifically, we propose a tiling scheme, which we call ClusTile, to minimize the amount of bandwidth needed to deliver viewed portions of 360-degree video segments. By "tiling scheme", we refer to a method of spatially slicing a projected 360-degree video segment to allow portions of the segment, rather than a single segment for the entire 360-degree view, to be transmitted over the network. Note that in these tiling schemes each tile represents a dynamic adaptive streaming (DASH) segment covering a portion of the 360-degree view and a (typically fixed) time interval, rather than a portion of a single 360-degree frame.

Tiling methods must address two tradeoffs to operate efficiently. First, as tile sizes decrease, video encoding efficiency decreases. On one hand, tiling methods must select tiles that are small enough to cover portions of 360-degree views without wasting too many pixels. On the other hand, tile sizes cannot be so small that the benefit of wasting fewer pixels is overtaken by decreased encoding efficiency. As an extreme example, imagine that a tiling scheme selects single-pixel tiles. These tiles could cover any user's view exactly, but if they were independently encoded, it would be as if no compression was applied.

The second tradeoff involves a choice between using larger numbers of tiles, thereby consuming larger amounts of storage space but allowing more efficient streaming of a given view, versus selecting fewer tiles for a segment, using less storage space but reducing the possible tiling options of a user's view during streaming. To illustrate this tradeoff, imagine that we store rendered 100- by 100-degree views at 10-degree intervals for all 10-degree buckets in the 360-degree sphere. This scheme would use more storage than the single equirectangular image, but would allow any individual view to be efficiently transmitted.

Our ClusTile method selects sets of tiles to encode and store at the server side by solving an integer linear program (ILP) formulated to capture an approximation of these tradeoffs. We derive the ClusTile ILP from a much larger ILP that more-directly represents the coding efficiency and storage tradeoffs but cannot be practically solved. This larger ILP attempts to efficiently cover views collected from a dataset of segment views. ClusTile also benefits from a clustering procedure designed to further reduce computational overhead. Rather than running a separate optimization problem on many individual views, clustering these views reduces the computation needed by solving a smaller number of optimization problems on clusters of views.

A potential disadvantage of ClusTile is that sets of tiles it produces may overlap, and, when streaming, there is no naive way to select tiles from this set that minimizes network bandwidth. We show, however, that a simple modification of our integer program produces a fast tile selection method that, in combination with caching, makes tile-selection practical during streaming.

Evaluation shows that under perfect prediction of user head orientation, ClusTile saves up to 76% downloaded volume compared to the baseline non-tiling scheme, and up to 52 % compared to best-performing fixed tiling schemes.

II. Background and Related Work

Unlike traditional video streaming, which encodes a view from a single perspective, 360-degree videos encode a full, omnidirectional view. To do so, 360-degree video frames are first projected onto a 2D plane before encoded using standard video codecs such as H.264 [26].

A. 360-degree Video - Projection and View Rendering

While many projection schemes exist, the equirectangular projection [2] is the most commonly used method for projecting 360-degree frames onto a 2D plane. This projection is currently used by 360-degree video streaming service providers such as YouTube and JauntVR.

In the equirectangular projection, pixels on the sphere are mapped to the 2D rectangular frame based on their yaw and pitch angle values (the yaw and pitch values of a point on a sphere are determined by applying the yaw motion first and pitch motion second). The yaw angles range from -180 degree to 180 degree from the left to right in the equirectangular image, while the pitch angles range from 90 degree to -90 degree from top to bottom. For example, a pixel at $< yaw = 0, pitch = 0 >$ on the sphere is projected to the center of the equirectangular projection.

During 360-degree video streaming, the view displayed to the user is rendered at runtime based on the user's head orientation (e.g., in Euler angle $< yaw, pitch, roll >$) and the field of view (FOV) (e.g. 100-degree vertical and horizontal). For every pixel in the destination image, we project a ray from the center of the sphere to the destination pixel. This ray will intersect with a point on the sphere. We can then fill this pixel value in the destination rendered image from the pixel at point where the ray described above intersects with the sphere's surface.

B. 360-degree Video Streaming

Today, 360-degree video streaming leverages bitrate adaptation provided by Dynamic Adaptive Streaming over HTTP (DASH) [18]. In DASH, videos are divided temporally into **segments**. A video segment is a sequence of video frames downloaded as a single unit during streaming. Segment durations can vary, but they typically consist of about one or a few seconds of video. Multiple versions of each segment are encoded at different bitrates. Over time, as available bandwidth fluctuates, the video streaming client will select a segment whose bitrate matches available network bandwidth. This selection procedure enables algorithms to flexibly tradeoff between reducing video quality and stalling (stopping playback to buffer additional video), allowing smooth playback under varying network conditions.

Bitrate adaptation, however, cannot address a fundamental inefficiency in 360-degree video streaming. That is, not all downloaded pixels in a video frame is consumed by the user. While 360-degree videos can provide a full 360-degree horizontal field of view, most VR headsets today have only between 90-degree to 110-degree horizontal FOV [9]. As a result, many of the pixels in a 360-degree video frame are not displayed. For example, to render a view of 100 degrees horizontal and vertical FOV centered at $< yaw = 0, pitch = 0 >$, only approximately 14.3% of the pixels in an equirectangular frame are needed. The network bandwidth used for downloading the rest 85% pixels is thus wasted.

To address this problem, one approach is to design an oriented projection scheme that devotes more pixels in the 360-degree video frame to the area on the sphere that matches the user's view orientation. One example of such scheme is the "offset cubic projection" proposed by Facebook. However, our measurement results have shown that this scheme suffers significantly from reduced encoding efficiency due to the distortion applied on the spherical surface [28]. As a result, even if this projection can produce similar visual qualities in less than half of the pixels compared to the equirectangular projection, the reduction in bitrate of the encoded video is only 5.6% to 16.4%. A further disadvantage of the offset cube projection is that a separate offset cube segment must be encoded for many orientations to support high qualities in all orientations. For example, Facebook uses 22 offset cube orientations [28] in their implementation. These many versions of the oriented projection significantly increase the storage space used to stream supported videos.

Another approach is tile-based streaming [14], [15], [16], [17], [19], [22], [23]. A tile is a rectangular area on the projected 360-degree video's frame. Tiles are independently encoded, allowing them to be decoded independently. Tile-based streaming is typically used with DASH, cutting a DASH segment into multiple tiles that jointly cover the entire frame area of the segment. In an ideal scenario, the video streaming client only need to download a minimum set of tiles that cover the user's view.

A challenging problem with tiling is in determining tile sizes that best support the streaming application. Small tile sizes can minimize the number of pixels transmitted over the network that are not observed by the user. However, as the number of pixels in a tile decreases, the video compression ratios also suffer [12], [21], [25]. These decreased compression ratios occur because motion estimation, the key to inter-frame compression in modern video codecs, is less efficient when the search range of block matches is constrained to small tiles. As a result, a tiling strategy that minimizes tile sizes may not minimize downloading bandwidth. On the other hand, while large tile sizes can improve compression ratios for individual tiles, these large tiles may result in larger numbers of unviewed pixels.

Existing tile-based streaming prototypes mainly use *fixed-tiling*: video frames are cut into non-overlapping tiles with size *tile_width* $\times$ *tile_height* each [4], [7], [15]. In prior work, we proposed a method called OpTile. OpTile optimizes an objective that considers both the server-side storage cost and bandwidth cost. The tiling solutions from this optimization problem are both non-overlapping and allow for variable sized tiles [27]. A downside of OpTile is its restriction to non-overlapping tiles, which limits bandwidth savings.

III. Design of ClusTile

The ClusTile objective was developed from the intuition that the most important aspect of 360-degree video streaming involves minimizing the bandwidth used to deliver video streams to the client. We make the assumption that there is some underlying distribution of views for each segment, and that this distribution can be effectively estimated by observing the pattern of past views of the segment. Our objective thus relies on a dataset of past views.

The solution to such an objective clearly involves selecting a tile corresponding to each past view so that this tile covers the view as precisely as possible. Equally clearly, the problem with this approach is that as the number of past views increase, the number of tiles will also increase so that all possible tiles that almost-exactly cover user views are included in the tiling solution. This solution, however, may cause practical problems, as the amount of storage needed for these tiles may increase past available system constraints.

We thus attempt to find a tiling solution that minimizes the required bandwidth with a set of tiles that also can be stored in a pre-defined amount of space. Adding this constraint unfortunately produces an ILP that is intractable to solve even for relatively small datasets, as the ILP must be solved jointly for all dataset items. A dataset of only ten items requires seven minutes of computation on commodity machines, and the solution scales polynomially in the number of dataset items.

TABLE I: Variables used in the problem formulation.

x	A vector representing the set of tiles selected and encoded for streaming.
c	A vector representing the downloading (i.e., bandwidth) cost of each tile.
M	A matrix mapping tiles (column) to basic tiles they cover (row).
$x^{(v)}$	A vector representing the set of tiles used to cover view v.
$b^{(v)}$	Basic tiles partially or fully covered by view v.
$x^{(k)}$	A vector representing the set of tiles needed to cover all views in a cluster k of views.
$b^{(k)}$	$b^{(k)}$ is a binary vector indicating which basic tiles are included in cluster k
$n^{(k)}$	A vector of counts of the number of views in cluster k that are covered by tiles indexed by each element of the vector. Given a vector $m^{(k)}$, where $m^{(k)}$ indicates the number of times a basic tile is covered, $n_j^{(k)} = \max_i(m_i^{(k)} \times M_{ij})$.

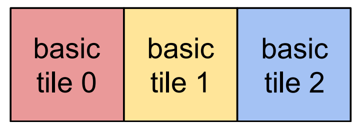


Fig. 1: Dividing a segment into three basic tiles.

To overcome this computational limitation, we propose to run our ILP over clusters of views in the dataset rather than over individual views. The speed increase results from the observation that clusters of views can be computed very quickly with standard tools compared to solutions to the ILP. In addition, if many views in our dataset are very similar, then it may not be useful computing tiling solutions multiple times. Grouping these similar views together is a natural way to address this inefficiency in computation as we expect similar views to produce similar tile coverage solutions.

A. Problem Formulation

To formulate the problem of finding the best tiling solution, we must first create a candidate set of tiles. To create this candidate set, we use the same method we proposed in OpTile [27]. We describe the method here so that the problem formulation is complete.

Given a video segment, we first construct a set of **basic tiles**. Each basic tile is the smallest tile that can be encoded and downloaded during video streaming. Any **valid tile** must be rectangle and exactly covers one or more basic tiles. For example, if we divide a video segment spatially into 3 basic tiles as in Figure 1, there exists a total of six possible tiles that can be created from these basic tiles, shown in Figure 2. We thus can use a binary vector x to encode the presence of these possible tiles in the solution: $[a, b, c, d, e, f]$. For example, if decision is made to cut the segment vertically into two disjoint tiles, *a* and *e*, then x will be represented as $[1, 0, 0, 0, 1, 0]$.

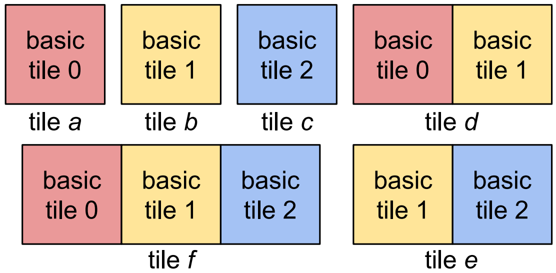


Fig. 2: A candidate set of six possible tiles can be constructed if the segment is divided into 3 basic tiles.

Different from OpTile [27] which considers only non-overlapping tiles, in ClusTile, it is possible to select overlapping tiles. For example, if $x = [0, 0, 0, 1, 1, 0]$, selecting tiles d and e, then basic tile 1 is covered twice.

We use a vector c to represent the amount of bandwidth needed to download each tile. This amount of bandwidth can either be estimated or computed directly by encoding the corresponding tile.

We formulate the following integer linear program (ILP) objective intended to find an optimal set of tiles used to stream a segment of 360-degree video. The objective is intended to minimize the expected network bandwidth used to stream a view of the segment, assuming fixed distribution over segment views, subject to a storage constraint. Here, the optimal set of tiles is encoded in the value of x.

Joint Bandwidth Minimization Objective:

$$\begin{aligned} \text{minimize:} \quad & \sum_v c^\top x^{(v)} \\ \text{subject to:} \quad & M x^{(v)} \geq b^{(v)} && \forall v \\ & x_i^{(v)} \in \{0, x_i\} && \forall v, i \\ & x_i \in \{0, 1\} && \forall i \\ & c^\top x \leq S \end{aligned}$$

In the *Joint Bandwidth Minimization Objective*, for a view v of the segment, $b^{(v)}$ is a binary vector where a 1 indicates that a basic tile is at least partially shown by the view. $x^{(v)}$ is a subset of tiles in x used to cover all basic tiles in $b^{(v)}$, as required by view v. This coverage constraint is expressed using a matrix M. M is a binary matrix whose rows represent basic tiles and whose columns represent tiles. A value of 1 at position (i, j) of M indicates that the tiles represented by column j cover the basic tile represented by row i. For example, the contents of M for the example in Figure 2 is shown below:

basic tile	a	b	c	d	e	f
0	1	0	0	1	0	1
1	0	1	0	1	1	1
2	0	0	1	0	1	1

The *Joint Bandwidth Minimization Objective* minimizes the bandwidth used in streaming, over an entire dataset of views, subject to a constraint on the total storage size of all tiles in x. However, there are two practical problems associated with this objective. First, to compute the cost vector, c, we need to know $O(n^2)$ tile costs, where n is the number of basic tiles. To do so, either the tiles can be encoded to compute these storage costs, at significant computational expense, or these costs can be estimated. Estimation requires that the estimation errors be small enough, or bound the true tile sizes closely enough, that the ILP solution will still have acceptable performance.

The second and more-difficult problem to surmount, is that the size of this integer linear program is proportional to the number of items in the dataset. These integer linear programs become infeasible to solve even for small datasets. For example, a dataset of ten views takes about 400 seconds to solve on commodity hardware.

We propose a tractable approximation of the *Joint Bandwidth Minimization Objective*. Solutions to this approximation can be computed in practical amounts of time on commodity hardware but may not achieve the precise guarantees of the original objective.

ClusTile Objective:

$$\begin{aligned} \text{minimize:} \quad & c^\top \text{diag}(n^{(k)}) x^{(k)} \\ \text{subject to:} \quad & M x^{(k)} \geq b^{(k)} && \forall k \\ & x_i^{(k)} \in \{0, 1\} && \forall k, i \\ & \sum_i x_i^{(k)} \leq N && \forall k \end{aligned}$$

The *ClusTile Objective* is similar to the *Joint Bandwidth Minimization Objective*, defining a comprehensive objective over a set of views. There are four main differences.

First, rather than attempting to cover individual views, it attempts to cover clusters of views. Here, the vector $b^{(k)}$ is a binary vector indicating which basic tiles are included in cluster k:

$$b^{(k)} = \bigvee_{v \in k} b^{(v)}$$

If any view in cluster k includes a basic tile, then we indicate this basic tile in $b^{(k)}$ by a 1 in the corresponding vector element. However, we note that other thresholds on cluster contents may be possible for constructing $b^{(k)}$. Specifically, basic tiles covered by a small number of cluster elements may be excluded from $b^{(k)}$.

Second, unlike the *Joint Bandwidth Minimization Objective*, the cost vector in the *ClusTile Objective* is scaled by the number of views in the cluster that are covered by a given tile, with this weighting given by the vector $n^{(k)}$.

Third, we include a constraint on each cluster's solution, x_k, requiring that there are no more than N non-zero items in x_k. That is, no more than N tiles should be used to render views required by cluster k. This constraint is based on the following practical concern: With H.264, each tile has to be independently decoded. This means for every tile needed for rendering the view, a decoder instance has to be initiated. All decoders have to complete their decoding tasks timely. This can result in high context switch overhead if there are too many instances. The decoded streams also have to be synchronized before view rendering can take place. Such synchronization overhead can be big if too many tiles are required for rendering a view.

Finally, the *ClusTile Objective* removes cross-view constraints, x, and the total storage size constraint, $c^\top x \leq S$. Removing these cross-view constraints allows us to solve the *ClusTile Objective* for each cluster independently, improving the tractibility of the ILP. The combined solution, x, for all clusters is constructed as $x = \bigvee_k x_k$. Despite the absence of the total tile size constraint, we expect solutions for all clusters to consume reasonable amounts of space as long as the number of clusters is limited.

The *ClusTile Objective* also improves computational tractability by reducing the integer linear program size through inclusion of multiple views into a single clustered view – clusters can be computed much more quickly on a per-view basis than ILP solutions. A potential disadvantage with this approach is if clusters have high variances, then the coverage computed for a cluster will result in wasted bandwidth when streaming a single view. That is, if clusters cannot tightly represent groups of views, then the optimal way of selecting tiles to cover a single view from the set provided by the clustering objective may also frequently cover unviewed areas of the 360-degree sphere, making this clustering objective inefficient. We attempt to evaluate the effect of this clustering approximation empirically in Section V.

As a final concern, we note that solutions to these objectives might lead to situations where we cannot cover views outside of our training set with tiles in the solution. That is, solutions to our objectives may not generalize well outside of the set of sampled views. A possible "regularization" strategy could involve always including the set of basic tiles in the solution. This approach would ensure that all views not seen in the training set were coverable but would come at the cost of extra storage. We use this approach in our experiments in Section V.

B. Cost Estimation

An important piece of our integer program formulation is the cost c associated with each item in x. Here, c_i represents the storage size of the segment associated with tile x_i. Given n basic tiles, there exists $O(n^2)$ solution tiles. This large number of tiles makes encoding each tile to obtain its size too computation intensive to be practical.

Instead, we estimate the storage size of each possible tile through a prediction strategy that takes the strong correlation between video compression and motion estimation into account. To do so, we use an approach we previously proposed in [27]. We describe the approach here so that this section is self-contained.

Video codecs today exploit inter-frame similarity for compression through block-based motion estimation. To encode a block, the video encoder searches for a best-matching block in nearby frames. The relative location of this best-matching block to the block being encoded is then recorded as a "motion vector". The video encoder then computes the difference between the current block and the best-matching block and compresses this difference (residual). If the current block and the reference block are highly similar, then only a small amount of data is needed to encode the residual. Cutting a frame into tiles reduces the population of potential block matches available to the video encoder. As a result, with smaller tiles, a less similar reference block may be located and used for motion estimation, leading to a larger-magnitude residual that cannot be compressed as efficiently.

TABLE II: Features used for tile size estimation.

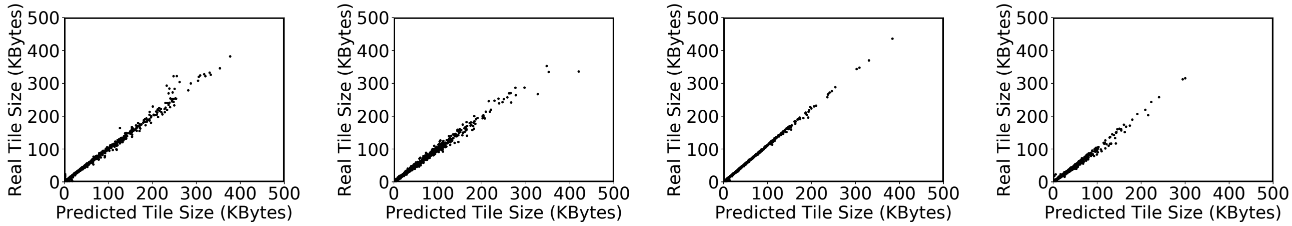
$\sum_{i \in t} s_i$	Total storage size of all basic tiles inside tile t if they are all encoded independently.
$\sum_{i \in t} mv_i$	Total number of motion vectors that need to be relocated if all basic tiles in tile t are encoded independently.
mv_t	The number of relocated motion vectors if tile t is encoded independently. Note that this set does not include the mv_i describing motion vectors between basic tiles found within tile t.
o	Average storage overhead of each relocated motion vectors in this segment.
n_t	Number of basic tiles inside t.

Our cost estimation is done on video segments. Each segment contains a short duration of video, typically one to three seconds, and can only contain one group of pictures (GOP). That is, each segment can contain one single intra-coded I-frame. We denote the storage size of the non-tiled, original segment as s. We then spatially cut this segment into basic tiles of selected size. Size of basic tiles must be multiples of the H.264 macroblock, i.e., 16×16. Our prediction method begins by encoding all basic tiles. We denote the i^{th} basic tiles storage size as s_i. The method then extracts all motion vectors in the segment and counts how many motion vectors pointing to a basic tile i would have to be relocated if i was encoded independently. We denote this count as mv_i. The prediction procedure then calculates the storage overhead per relocated motion vector as $o = \frac{\sum_i s_i - s}{\sum_i mv_i}$.

Given a tile t that contains more than one basic tiles, our prediction method estimates t's storage size using the five features shown in Table II. We trained an artificial neural network (ANN) model using a dataset of 6,082 tiles. To generate these tiles, we first cut the video temporally into one-second long segments. We then randomly select a segment from a video, a tile position, and tile width and height. Once a tile was selected, we extracted the features from Table II and encoded the tile using *ffmpeg* [3] to obtain its true tile size in bytes. We configured our ANN to use a single hidden layer of 50 nodes with the ReLU activation function. L-BFGS [20] was used for training the ANN.

Figure 3 shows prediction results from our four-fold cross validation. In each fold, we trained an ANN model based on features and empirical tile sizes from three videos. We then evaluated the trained model on tiles from the fourth video. The overall median absolute error in the prediction was 4.7%, varying from 2% to 8% over the four folds.

We used the ANN model trained with all four videos in the



(a) Video 1 (b) Video 2 (c) Video 3 (d) Video 4

Fig. 3: Results from four-fold cross validation. In each fold, we trained a neural network model based on features and empirical tile sizes from three videos. We then evaluated the trained model on tiles from the fourth video.

dataset for tile size (i.e., vector c) estimation for our evaluation in Section V. This set of four videos used for ANN training does not overlap with the set of five videos used for bandwidth reduction evaluation in Section V.

C. Streaming Client Tile Selection

Since each basic tile may be covered more than once by the tiles in x, a tile subset selection algorithm is required for selecting $x^{(v)}$ given x and $b^{(v)}$.

Client Tile Selection Objective:

$$\begin{aligned} \text{minimize:} \quad & {c^{(r)}}^\top x^{(v)} \\ \text{subject to:} \quad & Mx^{(v)} \geq b^{(v)} \\ & x_i^{(v)} \in \{0, x_i\} \quad \forall v, i \end{aligned}$$

For all i in x where x_i is 1, $c_i^{(r)}$ represents the true storage size of the tile i. $c^{(r)}$ is obtained empirically after all tiles in x have been encoded. This integer program then solves the optimal $x^{(v)}$ given x, M, $b^{(v)}$, and $c^{(r)}$. Since the number of non-zero items in x is typically small, we can reduce the elements of $x^{(v)}$ and columns of M to only those that correspond to non-zero elements of x, allowing the integer linear program to be solved very fast. We report the solving time in Section V.

IV. Implementation

For each video segment, we construct the *ClusTile Objective* using a known set of $d^{(v)}$s and estimate cost vector c using the ANN model. When constructing matrix M, we limit the height and width of each tile so that a tile can contain up to 12×12 basic tiles. We use an off-the-shelf implementation of K-means [8] to cluster views and use Gurobi Optimizer 7.0.2 [5] to solve integer linear programs.

ffmpeg is then used to crop and encode tiles based on the ClusTile solution. To maintain the same quality levels for all tiles, we use the same x264 parameters for encoding all cropped tiles. The same procedure is repeated for all segments in a video.

V. Evaluation

We compare ClusTile against two baseline 360-degree streaming solutions: i) `origin`, a solution where no tiling is employed and ii) `fix-n`, a fixed tiling scheme where segments are cut into fixed-size tiles, each tile encoding $n \times n$ pixels. We also compare ClusTile's bandwidth savings with our own prior work, `OpTile` [27].

Our ClusTile objective requires a set of existing views of each segment as well as the basic tile coverage, $b^{(v)}$, of these existing views. For this, we use a publicly-available dataset: the 360-degree video head movement dataset [13]. This dataset records the head orientation of 58 users watching 5 monoscopic 360-degree videos, encoded in the equirect-angular projection, in VR headsets. We downloaded these 5 videos and re-encoded them at two resolutions: 1920×960 and 3840×1920. For each video in the dataset, users watched each video for about 80 seconds. We extracted these 80 seconds from each video and encoded them into one-second segments.

After this segment extraction step, we then take each segment and construct a set of basic tiles. For the 1920×960-resolution videos, we set the basic tile to contain 4×4 H.264 macroblocks, i.e., 64×64 pixels. For 3840×1920 videos, we use larger basic tiles, 128×128 pixels, in order to reduce the number of solution tiles in x. These size selections mean that for both video resolutions, the number of basic tiles is $30 \times 15 = 450$.

Once basic tiles are produced, we randomly select 40 users from the dataset as a training set and extract their view orientations for each temporal segment. A user's view direction may change over the playback of one segment, e.g., the user may pan through the video within the one-second segment duration. To account for this, the head movement dataset records multiple instantaneous view orientation samples in one second. To construct $b^{(v)}$, we assume a viewport of 100-degree horizontal and 100-degree vertical field of view. For a user u's view v over a segment, if any pixel in a basic tile i is used when rendering any instantaneous views of user u in the same segment, we set $b_i^{(v)}$ to 1. We repeat this strategy to construct the set of 40 $b^{(v)}$s for every segment. We then cluster these views into K clusters. We experiment with two different settings of K: 5 and 10, creating 5 or 10 $b^{(k)}$s for the ClusTile objective. We refer to these settings as `ClusTile-5` and `ClusTile-10`. For each cluster k, we limit the maximum number of tiles needed to cover all basic tiles, $b^{(k)}$, in the cluster to be 10.

To construct vector c as required by our objective, for each segment, we estimate the storage size of each possible tile

TABLE III: Average number of tiles in ClusTile solutions.

	1920 × 960		3840 × 1920	
	mean	median	mean	median
ClusTile-5	46.3	50	47.7	50
ClusTile-10	85.4	89	91.4	95

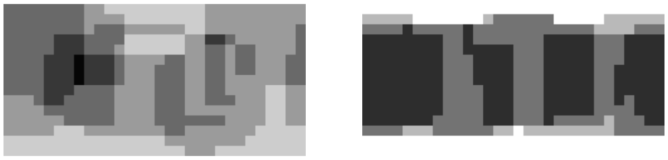


Fig. 4: Tiles generated by ClusTile-5 solutions. The darker an area is, the more times the corresponding basic tile is covered.

using the ANN model described in Section III-B.

After solving the ILP, we cut each segment into tiles based on tiles encoded in x. We encode each tile in x using *ffmpeg*. We construct the vector $c^{(r)}$ from these true encoded storage sizes to evaluate client-side tile selection time.

A. Server-side Tile Storage Size

Since we set the maximum number of tiles needed to render a cluster of view to 10, the maximum number of tiles in the ClusTile solution is $K \times 10$, where K is the number of clusters we use in our ClusTile objective. That is, `ClusTile-5` solutions can contain at most 50 tiles, and `ClusTile-10` solutions can contain at most 100 tiles. Table III shows the mean and median number of tiles in ClusTile solutions for videos in both resolutions. Figure 4 shows two examples of selected tiles in ClusTile-5 solutions. We use grayscale to indicate the number of times a basic tile in the segment has been covered by selected tiles. The darker an area is, the more times its corresponding basic tiles have been covered by tiles in the solution.

For every segment, we sum up all the tiles in the ClusTile solution and normalize the sum against the original, non-tiled, equirectangular segment. Figure 5 shows the results. For segments in both resolutions, the median normalized storage size of `ClusTile-5` is only 2.3. For `ClusTile-10`, since more tiles are generated, its median normalized storage size is slightly bigger, 3.6. If we apply the "regularization" strategy which includes all basic tiles in the solution, the normalized server-side storage size is increased by 2.1 for 1920 × 960 videos, and by 1.3 for 3840 × 1920 videos.

Overall, ClusTile uses 2.3 to 5.7 times the original storage size. This is much smaller compared to the offset cubic projection scheme used by Facebook. In their scheme, 22 offset cube orientations are used, requiring each segment to be encoded in 22 versions, which significantly increases the storage requirement at the server. Compared to the non-tiled, equirectangular segments, ClusTile trades increased server-side storage for much more efficient downloading. Next, we show the bandwidth reduction results.

B. Bandwidth Reduction During Streaming

Streaming bandwidth savings rely on predicting a user's future view and only download portions of the video segment that is needed for view rendering. Although view prediction is outside the scope of this paper, recent studies have shown that user's head movement is highly predictable in the near future [11], [24]. In this study, we evaluate ClusTile's performance under two different prediction schemes: perfect prediction and naive prediction. Since we have used the view orientations from 40 users to construct the ClusTile objective, we evaluate the network performance on the remaining 18 users.

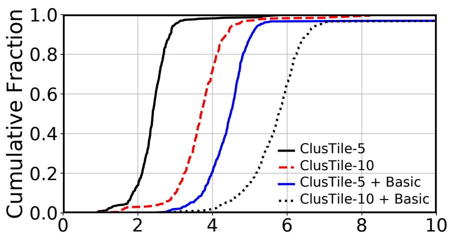
1) Perfect prediction: We first evaluate how ClusTile performs if we can perfectly predict how a segment is viewed by a user. Perfect prediction allows us to download the minimum subset of available tiles, with every tile used in view rendering.

The results are shown in Figure 6. Error bars in this figure represent the 95% confidence interval. `ClusTile-10` consistently performs the best, reducing 68% to 76% downloaded volume compared to the baseline, `origin`, where the full, non-tiled, equirectangular segment is downloaded. `ClusTile-5` performs slightly worse than `ClusTile-10`, saving 66% to 73% downloaded volume. Both `ClusTile-5` and `ClusTile-10` perform better than all `fix-n` schemes. `ClusTile-10` can save 19% to 52% downloaded volume over the best-performing `fix-n` scheme. Compared to `OpTile`[1], `ClusTile-10` can produce an additional 7% to 21% savings in downloaded data.

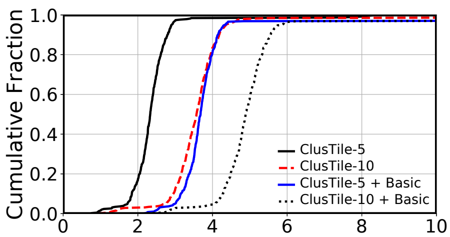
2) Naive prediction: We next evaluate the bandwidth efficiency when a naive prediction scheme is used for view prediction. In this naive prediction scheme, we assume that a user's view at second s will match his or her view in the interval $[s+3, s+4)$. Here, the view v_s at time s is an instantaneous view at the beginning of second s, while the a user may pan through the video in 1 second period. We further assume that perfect prediction can be made one second before a segment is due to play. Assuming that each segment is one second long, this perfect prediction will be able to accurately predict how segment $s+3$ is viewed, i.e., $v_{[s+3,s+4)}$ at time $s+2$. Thus at time $s+2$ in this naive prediction procedure, we allow additional tiles to be downloaded to cover basic tiles that were not downloaded at time s. That is, at time $s+2$ we download tiles at $b^{v_{\text{missing}}}$ where $b^{v_{\text{missing}}} = b^{v_{[s+3,s+4)}} \oplus (b^{v_{[s+3,s+4)}} \wedge b^{v_s})$. In this way, the total downloaded data of a segment is the size of tiles downloaded to cover b^{v_s} plus the size of tiles downloaded to additionally cover $b^{v_{\text{missing}}}$.

The results are shown in Figure 7. Across all videos and both resolutions, `ClusTile-10` consistently performs the best, saving 49% to 63% compared to `origin`. `ClusTile-5` performs the second best, saving 46% to 59% compared to `origin`. Compared to the best-performing fixed

[1]OpTile uses the parameter α to control the relative importance of storage and bandwidth costs [27]. Larger α yields more bandwidth savings. In [27], we tested OpTile with $\alpha = 0, 1,$ and 1000, and $\alpha = 1000$ yields the best results. We thus compare ClusTile with OpTile $\alpha = 1000$ in this paper.

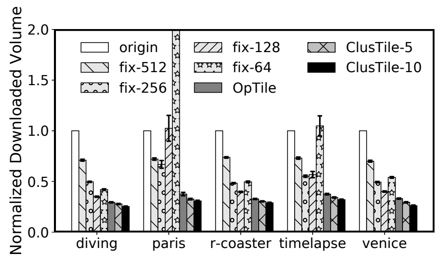


(a) 1920 × 960

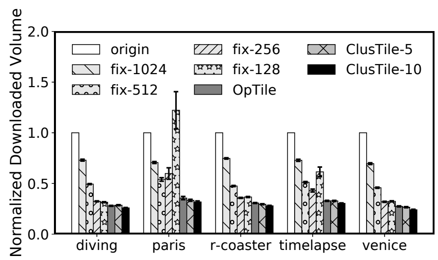


(b) 3840 × 1920

Fig. 5: CDF of normalized server-side storage size for various cutting schemes. In this figure, "+ Basic" indicates the normalized storage when all 450 basic tiles are also included in the solution.

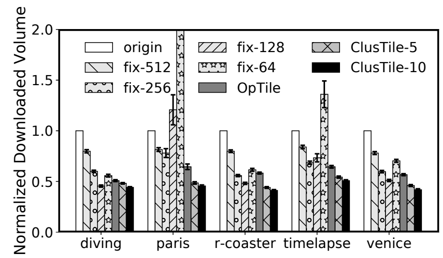


(a) 1920 × 960

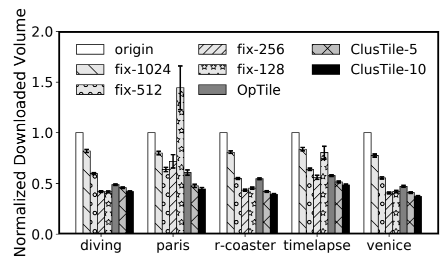


(b) 3840 × 1920

Fig. 6: Per-segment normalized downloaded volume averaged over traces from 18 users with perfect prediction.



(a) 1920 × 960



(b) 3840 × 1920

Fig. 7: Per-segment normalized downloaded volume averaged over traces from 18 users with naive prediction.

TABLE IV: Average number of tiles needed for rendering a segment.

	1920 × 960	3840 × 1920
ClusTile-5	6.8	7.6
ClusTile-10	7.0	8.1
fix-64	106.7	N/A
fix-128	32.2	106.7
fix-256	9.7	32.2
fix-512	2.2	9.7
fix-1024	N/A	2.2

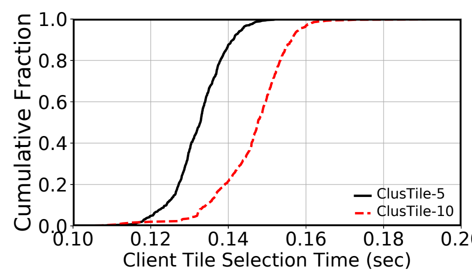


Fig. 8: Client tile selection time distribution.

tiling scheme, `ClusTile-10` performs at least as good ("diving" in 3840 × 1920) and can save up to 39% ("paris" in 1920 × 960) downloaded volume. Compared to `OpTile`, `ClusTile-5` can save 5% to 25% additional downloading volume, and `ClusTile-10` can save an additional 13% to 29%.

C. Client-side Tile Selection

We further show in Table IV the average number of tiles needed for rendering views required by a segment. Using tiles generated by ClusTile, on average, fewer than 9 tiles are needed. This is smaller than all but one fixed tiling scheme: `fix-512` for 1920 × 960 videos and `fix-1024` for 3840 × 1920 videos. `fix-512` and `fix-1024`, however, cut big tiles that require much more downloading bandwidth.

Finally, we report the solving time for the *Client Tile Selection Objective*. In our experiments, we used an Intel(R) Core(TM) i5-6600 3.30GHz CPU. Only one single CPU core is used for solving the ILP. The results are shown in Figure 8. The mean solving time for `ClusTile-5` is 0.132 seconds. For `ClusTile-10`, as the solution includes more tiles, the mean solving time is slightly longer, 0.146 seconds.

VI. Discussion and Future Work

In this section, we first discuss the strong dependence of our algorithm on data reflecting user behavior when viewing 360-degree videos. This strong dependence may lead us to consider what could go wrong if the data is missing or does not meet certain expectations. For example, how would ClusTile operate in the absence of any user data? Similarly, what if the distribution of views over a segment changes over time?

We expect that it should be possible to address either of these cases by simply recomputing the set of tiles, x, if we detect that the distribution of views over basic tiles has changed. These changes in distribution could be detected by applying a threshold on the distance between normalized vectors of basic tile view counts. The smaller this distance threshold, the more computation would be required, so the sensitivity of the tile re-computation process could be adjusted according to available resources.

For the case where no user data is available, ClusTile can select a fixed tiling scheme. After more empirical user views are collected and better tiles have been generated, these fixed tiles can be replaced.

Some significant difficulties in 360-degree video streaming using tiled segments involves strategies for predicting future views and strategies for best selecting tiles to download given these predictions. If there is a discrepency between tiles selected by a downloading stategy and tiles actually viewed by the user, then the ClusTile objective, which consumes historical user views may not be optimal. For example, if a user's view covers only basic tile 0 from Figure 2, but the client selects tiles a and b to download, then perhaps the server-side tile selection algorithm, e.g., ClusTile, should have included tile d, rather than tile a and b in the solution, as downloading tile d may be more efficient than downloading both tiles a and b. Future work involves better characterizing the sources of uncertainty during 360-degree streaming and developing tile selection algorithms to account for this.

VII. Conclusion

In this work, we propose ClusTile, a method to compute tilings of 360-degree videos. ClusTile is inspired by the need to minimize bandwidth used to stream 360-degree videos, assuming that users navigate through these videos in a way that can be characterized by recorded user behavior. ClusTile also attempts to constrain the amount of space needed to store tiles in its bandwidth-minimizing solution.

We first present an objective that jointly attempts to minimize bandwidth over past user views subject to a direct constraint on total tile storage space. Unfortunately, this objective is impractical to optimize on a large scale. We then improve the computational efficiency of our method by clustering user views and solving independent ILPs over these clusters. Taking the union of the solutions to these ILPs produces a set of tiles that can be served to efficiently render 360-degree segments during streaming.

This efficiency is demonstrated by our up to 76% bandwidth savings over non-tiled equirectangular segments and up to 52% savings over best-performing fixed tiling methods. For ClusTile-10, the best-performing ClusTile configuration, the storage space consumed by its solutions ranges from 3.0x of the non-tiled, equirectangular segment at the 10th-percentile and 4.3x at the 90th-percentile.

VIII. Acknowledgement

We appreciate constructive comments from anonymous referees. This work is partially supported by NSF under grant CNS-1618931.

References

[1] Akamai's [state of the internet] q1 2016 report. https://www.akamai.com/uk/en/multimedia/documents/state-of-the-internet/akamai-state-of-the-internet-report-q1-2016.pdf.

[2] Equirectangular Projection. http://mathworld.wolfram.com/EquirectangularProjection.html.

[3] FFmpeg. http://www.ffmpeg.org/.

[4] Gpac hevc tile-based adaptation guide. https://gpac.wp.imt.fr/2017/02/01/hevc-tile-based-adaptation-guide/.

[5] Gurobi. http://www.gurobi.com/index.

[6] Internet Connection Speed Recommendations. https://help.netflix.com/en/node/306.

[7] Kvazaar. https://github.com/ultravideo/kvazaar.

[8] sklearn.cluster.KMeans. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[9] Virtual reality headset comparison chart. https://freeflyvr.com/virtual-reality-headset-comparison-chart/.

[10] Virtual reality (vr) and 360 videos 101 - a beginner's guide. https://medium.com/visbit/virtual-reality-vr-and-360-videos-101-a-beginners-guide-70bbade8e39.

[11] 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 *2016 IEEE International Conference on Big Data (Big Data)*, pages 1161–1170, Dec 2016.

[12] C. Concolato, J. Le Feuvre, F. Denoual, E. Nassor, N. Ouedraogo, and J. Taquet. Adaptive streaming of hevc tiled videos using mpeg-dash. *IEEE Transactions on Circuits and Systems for Video Technology*, 2017.

[13] X. Corbillon, F. De Simone, and G. Simon. 360-degree video head movement dataset. In *ACM MMSys 2017*, 2017.

[14] L. D'Acunto, J. van den Berg, E. Thomas, and O. Niamut. Using mpeg dash srd for zoomable and navigable video. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 34. ACM, 2016.

[15] M. Graf, C. Timmerer, and C. Mueller. Towards bandwidth efficient adaptive streaming of omnidirectional video over http: Design, implementation, and evaluation. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, pages 261–271. ACM, 2017.

[16] M. Hosseini. View-aware tile-based adaptations in 360 virtual reality video streaming. In *Virtual Reality (VR), 2017 IEEE*, pages 423–424. IEEE, 2017.

[17] M. Hosseini and V. Swaminathan. Adaptive 360 vr video streaming based on mpeg-dash srd. In *Multimedia (ISM), 2016 IEEE International Symposium on*, pages 407–408. IEEE, 2016.

[18] ISO/IEC 23009-1:2014 Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats. Standard, International Organization for Standardization, May 2014.

[19] J. Le Feuvre and C. Concolato. Tiled-based adaptive streaming using mpeg-dash. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 41. ACM, 2016.

[20] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[21] K. Q. M. Ngo, R. Guntur, and W. T. Ooi. Adaptive encoding of zoomable video streams based on user access pattern. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 211–222. ACM, 2011.

[22] D. Ochi, Y. Kunita, K. Fujii, A. Kojima, S. Iwaki, and J. Hirose. Hmd viewing spherical video streaming system. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 763–764. ACM, 2014.

[23] D. Ochi, Y. Kunita, A. Kameda, A. Kojima, and S. Iwaki. Live streaming system for omnidirectional video. In *Virtual Reality (VR), 2015 IEEE*, pages 349–350. IEEE, 2015.

[24] F. Qian, L. Ji, B. Han, and V. Gopalakrishnan. Optimizing 360 video delivery over cellular networks. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 1–6. ACM, 2016.

[25] N. Quang Minh Khiem, G. Ravindra, A. Carlier, and W. T. Ooi. Supporting zoomable video streams with dynamic region-of-interest cropping. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 259–270. ACM, 2010.

[26] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[27] M. Xiao, C. Zhou, Y. Liu, and S. Chen. Optile: Toward optimal tiling in 360-degree video streaming. In *Proceedings of the 2017 ACM on Multimedia Conference*. ACM, 2017.

[28] C. Zhou, Z. Li, and Y. Liu. A Measurement Study of Oculus 360 Degree Video Streaming. In *Proceedings of the 8th International Conference on Multimedia Systems*. ACM, 2017.