OpTile: Toward Optimal Tiling in 360-degree Video Streaming

Mengbai Xiao*
George Mason University
mxiao3@gmu.edu

Chao Zhou*
SUNY Binghamton
czhou5@binghamton.edu

Yao Liu
SUNY Binghamton
yaoliu@binghamton.edu

Songqing Chen
George Mason University
sqchen@gmu.edu

ABSTRACT

360-degree videos are encoded for adaptive streaming by first projecting the spherical surface onto two-dimensional frames, then encoding these as standard video segments. During playback of these 360-degree videos, the video player renders the portion of the spherical surface in the direction of the user's view. These user viewports typically cover only a small portion of the 360 degree surface, causing much of the downloaded bandwidth to be wasted. Tile-based approaches can reduce the wasted bandwidth by cutting video spatially into motion-constrained rectangles. Streaming logic then only needs to download the tiles necessary to render the viewport seen by the user. Existing tile-based approaches cut 360-degree videos into tiles of fixed sizes. These fixed-size tiling approaches, however, suffer from reduced encoding efficiency. Tiling cuts away portions of the video that can be copied by the encoder from adjacent frames or within the current frame that are needed for effective video compression.

In this paper, we propose a scheme called OpTile. This scheme tiles a projected 360-degree segment by first estimating per-tile storage costs, then solving an integer linear program (ILP) to obtain an optimal, potentially non-uniform tiling. The ILP objective considers both content-specific characteristics and empirical distributions over user views of the segments. Using a randomly selected training/testing set split, we show that if a streaming algorithm can perfectly predict the user head orientation, our proposed scheme can save up to 73% of downloaded data compared to the non-tiling scheme and up to 44% compared to the best-performing uniform tiling methods.

ACM Reference Format:
Mengbai Xiao, Chao Zhou, Yao Liu, and Songqing Chen. 2017. OpTile: Toward Optimal Tiling in 360-degree Video Streaming. In *Proceedings of MM '17, Mountain View, CA, USA, October 23–27, 2017,* 9 pages.
https://doi.org/10.1145/3123266.3123339

*The first two authors made equal contributions to this paper.

MM '17, October 23–27, 2017, Mountain View, CA, USA

ACM ISBN 978-1-4503-4906-2/17/10…$15.00
https://doi.org/10.1145/3123266.3123339

1 INTRODUCTION

Virtual Reality (VR) devices, such as Samsung's GearVR and the HTC Vive, are moving increasingly toward the mainstream. These devices support a number of popular immersive experiences including interactive gaming environments, rendering and streaming "360-degree" videos. Unlike standard streaming video, 360 degree videos consist of frames that capture images fully surrounding a position in space. The VR device allows a user to select a view from the 360-degree by changing the orientation of his or her head and rendering the view at this chosen orientation. This mode of control simulates a real-life view of the surrounding environment.

To support 360-degree video streaming, video players typically download the entire encoded spherical view. Video players, however, often only render a small portion of the frame in the direction of the user's view. For example, viewports frequently encompass 100-degree horizontal by 100-degree vertical field of view (FOV). A FOV centered at $< yaw = 0, pitch = 0 >$ requires about 14.3% of all pixels encoded in the equirectangular frame.

This discrepancy between the downloaded and viewed data touches on a core problem in 360-degree video streaming: current production schemes download an entire 360-degree frame, but users consume only a small portion of this data.

This limitation of current streaming platforms has real-world consequences for potential 360-degree streaming users, potentially preventing wider adoption. Because i) VR devices must be placed in close proximity to the user's face, ii) the viewed portion of 360-degree videos does not take advantage of the full frame, and iii) stereo images halve the available resolution, the minimum quality needed to deliver VR users an adequate 360-degree video viewing experience is 4K. While the minimum recommended connection speed for viewing 4K resolution video is 25Mbps [7], the average broadband connection bandwidth in the USA is only 15.3 Mbps according to Akamai [2]. This could leave VR-360-degree video streaming out of reach for large populations of users.

Previously-proposed methods to address the 360-degree video wasted-bandwidth problem involve tile-based schemes. Here, 2D-projections of the 360-degree frame are broken up into rectangular tiles. These tiles are encoded as dynamic adaptive streaming (DASH) segments, and a subset of tiles from the full 360-degree segment can be served to a user. These schemes have the potential to significantly reduce the amount of bandwidth needed for 360-degree video streaming. However, they suffer from two serious drawbacks: i) tiling reduces the efficiency of video encoding. These methods encode frames by referencing areas of a past or future video frames. Tiling reduces the pool of such reference sub-images, reducing

encoding effectiveness. ii) the larger number of tiles increases the difficulty of the segment selection problem for streaming clients. Clients now must select a quality level for all tiles in a segment (including the possibility of a null quality level if a tile is not needed). In contrast, non-tile-based streaming clients must only select a single quality level.

In this work, we propose OpTile, a tile-based scheme that addresses the first general concern (reduced encoding efficiency) about tile based methods. We formulate an optimization problem (an integer linear program) that attempts to tile the 2D-projection of a 360-degree segment. The optimization problem presents a tradeoff between the costs of storing a set of tiles and the costs of serving sets of tiles that cover possible views of the segment. These views are weighted according to the distribution of the likelihood that they are selected by a user. Evaluation results show that under perfect prediction of user head orientation, OpTile can save up to 73% downloaded volume compared to the baseline non-tiling scheme, and up to 44% compared to best-performing fixed tiling schemes.

2 MOTIVATION AND RELATED WORK

In this section we explore the space of solutions for delivering 360-degree video streams. To do so, we also present background information needed to understand how these solutions operate.

2.1 Two-dimensional Video Encoding

Current 360-degree streaming systems all take advantage of the efficient encoding/decoding ecosystem on planar videos supplied by the families of popular coding standards including H.264 AVC [25] and HEVC [23] standards and their successors.

These standards encode videos using two primary steps. We present simplified versions of these steps to highlight areas relevant to this work. First for every "block"[1] in a frame image, the coding algorithm searches for similar blocks either within parts of the current frame that would be available to the decoder (that is, in already-decoded portions of the current frame) or in nearby frames that would be buffered by the decoder. When the encoder finds a block in a nearby frame that closely matches the current block, it encodes the position of this similar block in a "motion vector." Next, the encoder computes the difference between pixels in the current block and the reference block, and encodes this difference by applying a transform (for example, the discrete cosine transform), quantizing the transformed coefficients, and applying lossless entropy encoding (for example, Huffman coding) on the remaining sparse set of coefficients.

We describe these encoding steps because their efficiency can be impacted by how 360-degree video frames are represented. For example, tile based techniques can reduce the number of blocks available for copying (thus increasing the distance of the block match), and different 360-degree to planar projection can affect the sparsity of the coefficients output by coding transforms. Factors like these can reduce video encoding efficiency.

[1]The definition of "block" can vary across codec families, but similar concepts exist in most codecs.

2.2 Projecting 360-degree Surfaces onto the Plane

As methods for directly encoding 360-degree images and videos are not yet mature, 360-degree streaming systems currently take advantages of two-dimensional video encoding technology by first projecting the 360-degree spherical surface onto a rectangular image.

Projections used most frequently today are the equirectangular [3] and cubic projections [22]. The equirectangular projection is both the simplest and currently the most popular, used by most 360-degree video streaming service providers including YouTube.

With **equirectangular projection**, pixels on the spherical surface are mapped to the rectangular surface based on their yaw and pitch angle values on the sphere (the yaw and pitch values of a point on a sphere are determined by applying the yaw motion first and the pitch motion second). Given an equirectangular image with size of $width \times height$, a pixel on the rectangular surface, (x, y), is projected from a pixel on the spherical surface with $< yaw = (x/width - 0.5) \times 360, pitch = (0.5 - y/height) \times 180 >$. For example, a pixel at the center of the equirectangular surface is projected from $< yaw = 0, pitch = 0 >$ on the sphere.

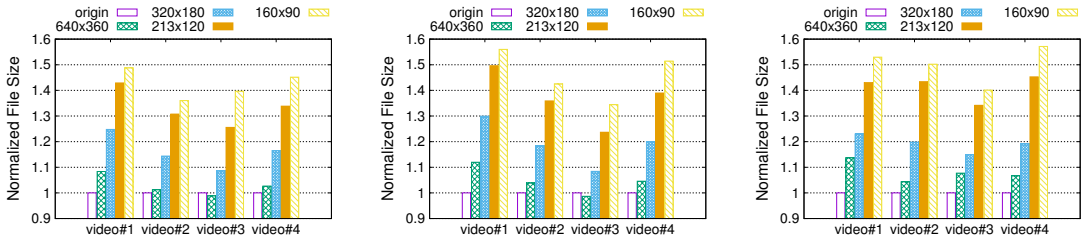
To create a **cubic projection** of a spherical surface, we first position the sphere within a cube. Rays are then projected outward from the center of the sphere. Each ray intersects with both a location on the spherical surface and a location on a cube face. We construct a cubic projection by taking the ray associated with a pixel on a cube face and finding the corresponding pixel on the spherical surface. For example, a ray pointing to the pixel at the center of the cube's front face intersects with the sphere at $< yaw = 0, pitch = 0 >$, so we would copy the pixel value at $< yaw = 0, pitch = 0 >$ to this front-face pixel. For encoding, the six cube faces are arranged on a planar image and compressed using standard video compression techniques. The cubic projection is used by Facebook [10].

2.3 360-degree Video Streaming

After projecting the sphere to a two-dimensional image, Dynamic Adaptive Streaming over HTTP (DASH) [18] can be supported as it is for standard videos. Specifically, the full sequence of two-dimensional frames is partitioned into temporal segments, and these segments are then encoded at multiple bitrates. During playback, the video player adaptively selects the quality level of video segments as playback progresses.

As mentioned earlier, directly serving equirectangular or cubic projected frames through DASH can waste significant amounts of network bandwidth. Projected frames encode the full spherical surface but a user's field of view (FOV) typically covers only a small portion of this surface. This discrepancy means a significant amount of transferred frame data is often never rendered. For example, a viewport of 100 degrees horizontal and vertical FOV centered at $< yaw = 0, pitch = 0 >$ covers approximately 14.3% of the pixels in an equirectangular frame.

Two main families of techniques have been proposed to address this inefficiency. The first family of approaches encodes the full spherical surface in each segment but quality is decreased in regions outside of an expected field of view. We call this family of approaches "oriented projections." The second group of approaches



(a) 720p (b) 1080p (c) 4K

Figure 1: Increase in video file size due to fixed tiling.

splits the 360-degree view into spatial segments, tiles, and allows each of these tiles to be downloaded at varying qualities. We call these "tile-based" approaches.

2.3.1 Oriented Projections. The pyramidal projection [9] and the offset cubic projection [28], both proposed by Facebook, are examples of schemes that encode the full 360-degree surface but vary the quality over this surface to reduce the amount of bandwidth devoted to regions that a user is unlikely to view.

In the pyramidal projection, the sphere is wrapped in a pyramid in a similar manner to the cubic projection's construction. Multiple pyramids are encoded, with each pyramid oriented so that the base is directed toward an expected viewing location. The pyramid geometry ensures that more pixels are devoted to a given angular area in the direction of the pyramid base.

The idea of the offset cubic projection is similar to the pyramid projection. The offset cube distorts the spherical surface to concentrate pixels toward a given orientation. The front face of the cube is then directed toward this orientation. Recently, Zhou et al. reverse-engineered the offset cubic projection used by Facebook for Oculus VR streaming [28]. The authors show that as long as the user's head direction is within 40 degrees of the offset cube orientation, views can be rendered with qualities as good or better than an equirectangular projection with more than twice the pixel resolution.

Despite this improvement in pixel-level efficiency, these approaches have some significant disadvantages:

Increased storage. Offset cubes and pyramids are oriented toward specific directions. To produce good visual quality for all possible user's head directions, many such orientations must be stored. For example, Facebook encodes 22 offset cube orientations for each segment. Given that Facebook also uses four quality levels, a total of $4 \times 22 = 88$ versions of the same video must be encoded and stored. This incurs significant storage overhead both at the datacenter and within CDN and proxy servers.

Reduced encoding efficiency. Although the offset cubic projection can significantly improve pixel-level efficiency, this efficiency does not always translate to equivalent levels of bandwidth efficiency. For example, Zhou et al. have shown that encoded offset cubic segments that contain less than half pixels than the equirectangular projection can only result in 5.6% to 16.4% average savings in video bitrate [28]. These poor compression ratios are likely a result of the distortion applied to the spherical surface. This distortion could cause motion estimation, the key to inter-frame compression, less effective.

2.3.2 Tile-based Approaches. In *tile-based* approaches, the two-dimensional projection is divided spatially into rectangular tiles. Tiles are then independently encoded and can be downloaded individually by a streaming algorithm. During streaming, the video player downloads tiles so that the user's predicted viewports over the segment duration are covered by high-quality tile-segments [12]. Tiles in regions that the user is not expected to observe can be downloaded at lower qualities, if desired, to account for unexpected changes in view orientations by the user.

Existing tile-based streaming prototypes mainly use *fixed-tiling*: video frames are cut evenly into $m \times n$ uniform tiles, or cut into tiles with size $width\_split \times height\_split$ each [6, 8]. Temporal segments in tile based schemes include a fixed number of frames, as in standard DASH implementations. Recently, Spatial Representation Description (SRD) [21] has been proposed as an amendment to the MPEG-DASH standard. SRD can be used to describe the spatial relationship among tiles that belong to the same temporal video segment within the DASH MPD file. This update in the MPEG standard has been used by many tile-based streaming schemes [13–17, 19] and is expected to make tile-based streaming more practical for deployment in production streaming systems. Besides DASH, Nasrabadi et al. proposed to use Layered Video Coding (LVC) to encode tiles in different quality levels [24].

Tile-based approaches can improve efficiency over both oriented projections and basic, un-oriented schemes: i) There is no need to store segments for multiple user orientations. ii) Tile based schemes need only download tiles covering the users' viewport.

However, tile-based schemes are impeded by **reduced encoding efficiency**. Splitting frames into tiles can reduce video coding efficiency, causing larger overall storage requirements, and potentially larger bandwidth requirements, in certain circumstances, than standard projections. This reduction in encoding efficiency occurs because motion vectors that reference the best block matches in the full segment can be cut by tile boundaries.

To understand the impact of tiling on video encoding efficiency, we selected four monoscopic 360 degree videos from YouTube. For each video, we prepared three versions with different resolutions: 720p (1280×720), 1080p (1920×1080) and 4K (3840×2160). To guarantee the same encoding parameters, we used *FFmpeg* [4] to re-encode all videos. We also removed the audio stream. We cut the videos spatially into tiles of fixed resolutions of 640×360, 320×180, 213×120, and 160×90. We then compared the resulting tile sizes with the size of original, non-tiled videos. The results are shown in Figure 1. For all four testing videos, video sizes become larger as the

number of tile-pixels decreases. The same trend exists regardless of video resolution. This trend is both reasonable and expected. Smaller tile size means more "cuts", leading to more sub-optimal motion vectors and residuals and less efficient video compression.

Focusing on the encoding efficiency, Zare et al. suggested using motion-constrained tile sets (MCTS) [27]. An MCTS contains all tiles in a single column. Within an MCTS, tiles in the top and bottom rows can be predicted from the tile in the middle. Since MCTSs are cut at fixed positions, content-specific characteristics and typical view patterns are not taken into account. Yu et al. proposed to split equirectangular representation horizontally into tiles and formulate a multi-dimensional, multiple-choice knapsack problem to decide the resolution and bitrate for each horizontal tile [26]. However, due to the high computation complexity, only the first video frame is considered in the problem formulation. In addition, Yu et al.'s formulation does not take into account user behavior that would make efficient encoding of one portion of the 360-degree view more important than another.

3 DESIGN OF OPTILE

We propose OpTile, a tile-based approach to the 360-degree streaming problem. OpTile attempts to address some of the shortcomings of the fixed tiling approaches. Namely, that fixed tiling schemes increase the storage space per pixel of the 360-degree view. As mentioned earlier, this encoding inefficiency occurs as a result of reducing the available blocks that can be copied compared to the full segment.

Intuitively, compared to a fixed tiling scheme, we would like to be able to increase some of the tile sizes so that the segments associated with these tiles can capture similar blocks needed for efficient coding. However, we would still like tiles to split the 360-degree frame so that it is possible to avoid transferring unviewed portions of the segment. Understanding where these splits would be most useful requires some information over the basic 360-degree setting. To understand which spatial areas of a segment can be efficiently encoded independently, we need information about the storage sizes of tiles with different dimensions. To understand where it is best to cut the 360-degree frame, we need information about user preferences of sequences of viewports across the segment.

We capture these competing concerns of encoding efficiency versus wasted data in an integer programming objective that considers a distribution over all possible views of a segment. Each possible view of the segment can be covered by different combinations of tiles. Our objective chooses a single covering of tiles for a segment that minimizes the total transferred bandwidth over this distribution of views over a fixed time period.

A separate portion of the objective considers the cost of storing the representation over this fixed time period. This storage portion of the objective competes with the downloaded bandwidth portion of the objective. For example, if an unpopular video is viewed only once in a year, then we would prefer a compact representation where we could expect to send the user more unviewed pixels.

3.1 Problem Formulation

Table 1 defines a few terms and variables used in our problem formulation. To aid in describing the problem we first present a sample use case. For this case, we partition a rectangular segment into four basic sub-rectangles. These basic sub-rectangles are arranged and indexed as shown in Figure 2.

Table 1: Definition of terms and variables used in this paper.

segment	A contiguous temporal subset of a video that can be downloaded as a single unit in dynamic adaptive streaming.
basic sub-rectangle	The smallest spatial division of a segment that can be downloaded during video streaming. Segments are partitioned spatially into rectangles to allow finer spatial granularity when downloading portions of a 360 degree view.
solution sub-rectangle	Any rectangular portion of the segment that can be constructed of one or more basic sub-rectangles.
x	A binary vector indicating the presence of a sub-rectangle in the solution.
$c^{(stor)}$	A vector with cost associated with storing each sub-rectangle.
$c^{(view)}$	Given a distribution over user viewports in a segment, an element of $c^{(view)}$ is the *expected* downloaded bytes of the associated sub-rectangle.
α	Weight assigned to $c^{(view)}$ in order to control the relative cost of storage compared to transferring a segment. It also accounts for the popularity of a segment.

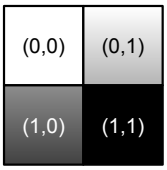


Figure 2: One rectangular segment partitioned into 4 basic sub-rectangles.

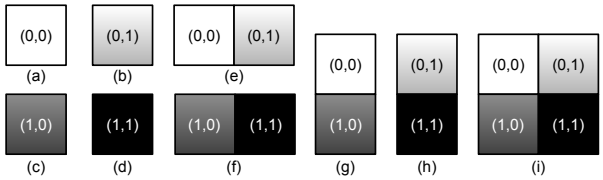


Figure 3: All 9 possible sub-rectangles that can be constructed if we partition a rectangle into two-by-two basic sub-rectangles.

Figure 3 shows the nine solution sub-rectangles can occur within this two-by-two rectangle: four - 1×1, two - 2×1 , two - 1×2, one - 2×2. We represent the presence of each of these rectangles in our solution with a binary vector x as follows:

$$[1 \times 1 \text{ at } (0,0), \quad 1 \times 1 \text{ at } (0,1), \quad 1 \times 1 \text{ at } (1,0),$$
$$1 \times 1 \text{ at } (1,1), \quad 1 \times 2 \text{ at } (0,0), \quad 1 \times 2 \text{ at } (1,0),$$
$$2 \times 1 \text{ at } (0,0), \quad 2 \times 1 \text{ at } (0,1), \quad 2 \times 2 \text{ at } (0,0)] \ ,$$

where "at (0,0)" indicates that the top left corner of the rectangle is located at basic sub-rectangle (0,0).

For x to be valid, every basic sub-rectangle must be covered exactly once by sub-rectangles encoded in x. For example, [0, 0, 0, 0, 1, 1, 0, 0, 0] is valid, including tiles e and g in the solution. On the other hand, [0, 0, 0, 1, 1, 1, 0, 0, 0] is not valid as basic sub-rectangle (1,1) is covered twice in this solution. [0, 0, 0, 1, 1, 0, 0, 0, 0] is also not valid as basic sub-rectangle (1,0) is not covered.

We have a storage cost vector, $c^{(stor)}$. This vector is the same length as the binary vector representing our solution. Each element of this vector is an estimate of the cost incurred including the corresponding sub-rectangle in the solution.

We also have to account for a network bandwidth cost for delivering a sub-rectangle. To do so, we need to account for all possible views of the 360-degree surface that can be delivered. To simplify the problem, we discretize all possible views of a segment into a set of size V. Each element of the set represents the event that a unique subset of basic sub-rectangles were displayed to a user who viewed a segment of 360-degree video. Note that the area of a video viewed in a segment can encompass areas from more than one viewing angle. For example, if the segment duration is one second, then a user could pan across this one-second of content. We associate a probability with each of the V segment view elements, $[p_1, \ldots, p_V]$. We consider also the cost of downloading a view, v, for a given solution, as the amount of data that needs to be downloaded for this view. This quantity can be expressed as

$$x^\top \text{diag}(d_v) c^{(stor)} \tag{1}$$

d_v is a binary vector that selects all sub-rectangles (according to the representation scheme described above for vector x) that cover the view, v. For example, if the two-by-two rectangle described above represents an equirectangular encoding of the 360 degree sphere, and we have a view at $< yaw = 0, pitch = 90 >$ - top of the equirectangular image, then the vector $d_{view-(0,90)}$ will be a length 9 vector: [1, 1, 0, 0, 1, 0, 1, 1, 1], indicating tiles a, b, e, g, h, and i contain basic sub-rectangles required for rendering the view.

Equation 1 thus can be interpreted as first selecting the subset of sub-rectangle costs covered by the segment view, v, then selecting a further subset of costs given a solution x.

Given a distribution over user viewports in a segment, an element of $c^{(view)}$ is the *expected* downloaded bytes of the associated sub-rectangle.

$$c^{(view)} = \sum_v p_v \text{diag}(d_v) c^{(stor)} \tag{2}$$

Finally, we encode the basic sub-rectangle coverage constraints for the optimization problem in a matrix, A. A is a binary matrix whose columns hold information about the basic sub-rectangles that a given solution sub-rectangle covers. For a two-by-two segment rectangle, A has four rows ($width \times height$ of the segment rectangle) and nine columns (the number of solution sub-rectangles). The contents of A for the 2×2 example is shown below:

Position	a	b	c	d	e	f	g	h	i
(0,0)	1	0	0	0	1	0	1	0	1
(0,1)	0	1	0	0	1	0	0	1	1
(1,0)	0	0	1	0	0	1	1	0	1
(1,1)	0	0	0	1	0	1	0	1	1

We construct an integer linear program (ILP) for this problem as follows:

$$\begin{aligned} \text{maximize:} \quad & (-c^{(stor)} - \alpha c^{(view)})^\top x \\ \text{subject to:} \quad & Ax = \mathbf{1} \\ & x_i \in \{0, 1\} \quad \forall i \end{aligned}$$

where $\mathbf{1}$ is a vector of 1's with 4 (the number of basic sub-rectangle) elements. The storage cost $c^{(stor)}$ can be interpreted as the monetary cost required for storing the sub-rectangles of a segment over a time interval Δt. The view downloading cost $c^{(view)}$ can be interpreted as the monetary cost required for transferring all sub-rectangles required for a view. α controls the relative cost of storage compared to transferring a segment. It should also account for the popularity of a segment. That is, α should be proportional to the number of times we expect the segment to be downloaded in the time interval Δt. We expect α can be estimated with reasonable accuracy from empirical measurements.

A linear program (LP) can be constructed by relaxing the constraint $x_i \in \{0, 1\} \forall i$ to $0 \leq x_i \leq 1 \forall i$. Solutions to the linear program are integral for all our experimental setups. These solutions can be computed for an x with 33,516 variables in 7 to 10 seconds on a single CPU core. Please see Section 4 for details.

3.2 Cost Vector Construction

To construct the integer linear program, we need to first construct the storage cost vector, $c^{(stor)}$. However, there exists $O(n^2)$ sub-rectangles, where n is the number of basic sub-rectangles. Therefore, it is not feasible to encode every sub-rectangle to obtain the storage cost. To this end, we propose to exploit the strong correlation between video compression and motion estimation to predict the values of $c^{(stor)}$.

Given a video, we first temporally divide it into segments of one second long each. Each segment is set to include only 1 GOP. We refer the size of a segment as S_{orig}. We then extract all motion vectors in every segment for later analysis. We then cut the segments spatially into basic sub-rectangles. Each such basic sub-rectangles include $4 \times 4 = 16$ macroblocks (i.e., 64×64 pixels). We encode each basic sub-rectangles independently and refer the size of each basic sub-rectangle as S_i. By analyzing the motion vector information, we can infer how many original motion vectors pointing into a basic sub-rectangle i should be relocated if this basic sub-rectangle i is encoded independently. We denote this as r_i. We calculate the storage overhead per motion vector as: $o = \frac{\sum_i S_i - S_{orig}}{\sum_i r_i}$, the overall increase in storage divided by the number of motion vectors intersected by the basic sub-rectangle boundaries. If basic sub-rectangles are "merged" into a bigger sub-rectangle, t, we use $m_t = \sum_{i \in t} r_i - r_t$ to denote the number of motion vectors that no longer need to be relocated due to the merge operation, where $i \in t$ indicates basic sub-rectangle i resides in t.

To estimate the size of an arbitrary sub-rectangle t, we used the following five features: $\sum_{i \in t} S_i$, $\sum_{i \in t} r_i$, m_t, o, and n, the number of basic sub-rectangles inside t.

We created a tile size dataset of 6,082 samples from four monoscopic 360-degree videos. Each of these four videos are encoded in two resolutions: 1920×960 and 3980×1920. To generate a tile, we randomly selected a segment from a video, a tile position (encoded

Table 2: MLP-based tile size prediction: results of four-fold cross validation.

Video	R^2	median absolute error (%)
Video 1	0.993	2.91
Video 2	0.987	5.09
Video 3	0.984	8.36
Video 4	0.992	3.65
Overall	0.989	4.72

as the left top position of the tile), and the tile width and height. We set a maximum tile size of 12×12 basic sub-rectangles. For each selected tile, we constructed a dataset element by first calculating the five-element feature vector, then encoded the tile's video segment using *FFmpeg* to obtain the space needed to store this segment.

We use a multi-layer perceptron (MLP) [11] (also known as an artificial neural network, ANN) approach to estimate tile sizes. Our MLPs were configured to include a single 50-node hidden layer with the ReLU activation function. Training proceeded for 300 iterations using L-BFGS [20].

To evaluate the MLP-based prediction, we use four-fold cross-validation. In each fold, we train the MLP on tiles from three videos and use the trained model to predict tile sizes of the fourth video. Table 2 shows the R^2 and the median absolute error (in %) results. Over all four folds, the median absolute prediction error is smaller than 5%.

4 IMPLEMENTATION

Figure 4 shows the overall workflow of OpTile. It first temporally divides the video into 1-second segments. For each segment, we solve an integer linear program to determine the optimal tiling strategy.

To construct the ILP, we estimate the storage costs of each tile, $c^{(stor)}$, using our neural network model, as well as a known set of views, d, and their probability distributions, p, to estimate the view downloading cost $c^{(view)}$. When constructing matrix A, we limit the maximum tile size to 12×12 basic sub-rectangles. For example, if we set the basic sub-rectangle to contain 64×64 pixels, then the maximum allowed tile size is 768×768 pixels.

To solve the integer program, we use the GNU Linear Programming Kit (GLPK) solver 4.61 [5]. We encode all possible solution sub-rectangles (i.e., all tiles not bigger than the maximum allowed tile size) into a binary vector x, representing the solution. During our experiments, we used a desktop machine with an Intel(R) Core(TM) i5-6600 3.30GHz CPU. GLPK can compute the optimal solution for a vector x with 33,516 variables in 7 to 10 seconds on a single CPU core.

The binary solution produced by GLPK indicates whether a possible sub-rectangle should be included in the solution. Based on this solution, we divide the segment spatially into tiles and use FFmpeg to encode these tiles with the same x264 encoding parameters. The same procedure is repeated for all temporal segments of a video.

Table 3: Average number of tiles in a segment per video.

Resolution: 1920×960

Video	OpTile			fixed cut			
	α=0	α=1	α=1000	fix64	fix128	fix256	fix512
diving	7	31	87	450	105	21	3
paris	6	17	50	450	105	21	3
r-coaster	6	17	50	450	105	21	3
time	6	19	59	450	105	21	3
venice	7	19	57	450	105	21	3

Resolution: 3840×1920

Video	OpTile			fixed cut			
	α=0	α=1	α=1000	fix128	fix256	fix512	fix1024
diving	6	49	149	450	105	21	3
paris	6	17	50	450	105	21	3
r-coaster	7	32	93	450	105	21	3
time	7	34	103	450	105	21	3
venice	7	41	112	450	105	21	3

5 EVALUATION

We compare OpTile against a variety of fixed-tiling strategies as well as against the non-tiled, full equirectangular projection. All tiling strategies use motion constrained tiling that can reduce the video compression efficiency. We focus on the following two evaluation metrics: (i) server-side storage requirements and (ii) the number of bytes downloaded by the streaming client.

To generate p_v and d_v as required by the integer program, we use user head movement data from the 360-degree videos head movements dataset [1]. This dataset contains head movement data of 58 users watching five monoscopic 360-degree videos in head-mounted displays. We download the five videos used in the tests and extract the corresponding portion of the video users watched during the tests, e.g., 80 seconds from each video. These videos are encoded using the equirectangular projection. For each video, we consider two resolutions: 1960×960 and 3840×1920. For videos in 1920×960, we set the basic sub-rectangle to contain 64×64 pixels. For videos in 3840×1920, we set the basic sub-rectangle size to 128×128 pixels in order to limit the number of variables in the integer program. We temporally segment these videos into segments of 1-second long each. For each temporal segment in a video, we generate the features required for the neural network model, i.e., $<\sum_{i \in t} S_i, \sum_{i \in t} r_i, m_t, o$, and $n>$, and use these features to predict storage sizes of all possible tiles, i.e., $c^{(stor)}$.

We randomly select a set of 40 users from the dataset. To generate the view distribution, we assume 100-degree horizontal and vertical FOV and use the head orientation of these 40 users to generate p_v and d_v for each temporal segment. That is, our tiling decision is made based on content characteristics of each segment as well as the empirical view pattern of 40 users.

We set $\alpha = 0, 1$, and 1000 in our integer programming objective to evaluate the performance as the view downloading cost takes different weights in the optimization. We then cut each temporal segment spatially based on the integer program's solution. For comparison, we also cut the segments into fixed tile sizes. We refer to these cutting schemes as "fixed cut". We consider a total of four "fixed cut" schemes. For videos in the resolution of 1920×960, tile sizes are 64×64, 128×128, 256×256, and 512×512. We refer to these fixed tiling schemes as *fix-64*, *fix-128*, *fix-256*, and *fix-512*, respectively. For videos in the resolution of 3840×1920, we cut

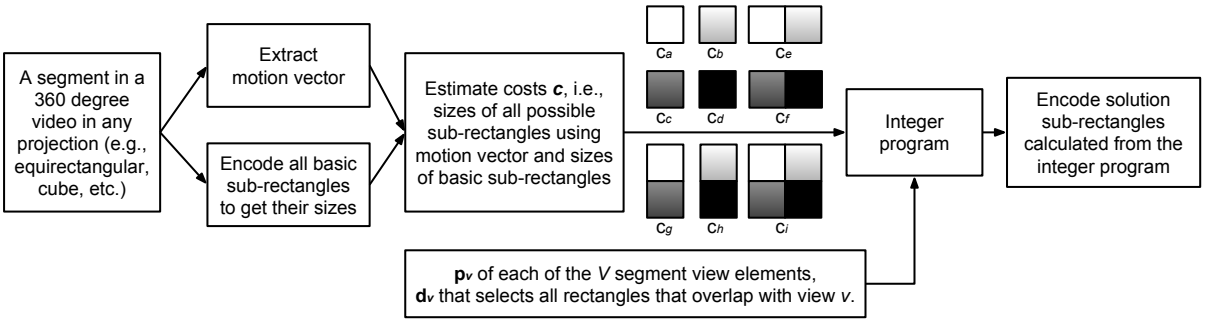


Figure 4: Overall workflow of OpTile, our proposed optimal 360-degree video tiling system.

them into 128×128, 256×256, 512×512, and 1024×1024 tiles. Table 3 shows the average number of tiles in a segment of a video as cut by OpTile as well as fixed tiling schemes. Note here that since we constrain the maximum tile size to 12 × 12 basic sub-rectangles, OpTile cuts a segment into at least 6 tiles.

5.1 Server-side Storage Size

We first evaluate the storage volume required for different cutting schemes. For each temporal segment, we sum up the storage sizes of all its tiles and compare this total size against the original, non-tiled segment size. The results are shown in Figure 5. The x-axis represents different videos and the y-axis shows the mean of normalized storage size of all segments in a video. Columns with different patterns indicate different tiling schemes. In this figure, *origin* means the original, non-tiled scheme, $\alpha = m$ represents our OpTile schemes when setting the weight of view downloading cost, $c^{(view)}$, as m, and *fix-n* is the fixed tiling scheme. Error bars show the 95% confidence interval.

We can see that across all videos in both resolutions, OpTile with $\alpha = 0$ results in roughly the same video size as the original, non-tiled video segment. This scheme sometimes leads to slightly smaller segment sizes than the original segment due to lossy compression during tiling. Note that since all tiling operations are conducted using the same video encoding parameters in both our schemes and the standard solutions, the lossy compression introduced due to re-encoding does not affect the fairness of our comparison.

If we set the weight of view downloading cost α higher, to 1000, the storage size of our scheme slightly increases. Storage size of fixed cutting schemes also increases as tile sizes get smaller. For videos in 1960 × 960, OpTile always requires smaller storage than fixed 64 × 64 and 128 × 128 schemes. For videos in 3820 × 1920, OpTile always requires smaller storage than fixed 128 × 128 and 256 × 256 schemes.

5.2 Client-side Downloaded Volume

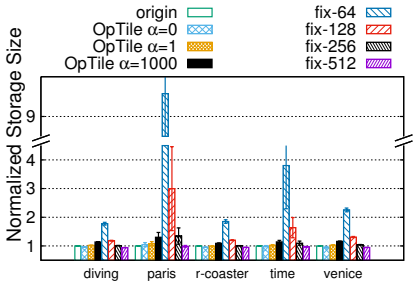
Next, we compare the number of bytes downloaded by the client when tiles are cut using our OpTile schemes versus using the fixed cutting schemes. Since we have used view distributions from a randomly selected set of 40 users to construct the integer programming problem, we use the remaining 18 users in the dataset to evaluate the network performance.

5.2.1 Perfect Prediction. For both our OpTile schemes and the fixed cutting schemes, we first assume that the client can make perfect prediction of user's head orientation and only download the required tiles. That is, we evaluate the client's downloading volume when no downloaded tiles are wasted (i.e., downloaded but not rendered during playback).

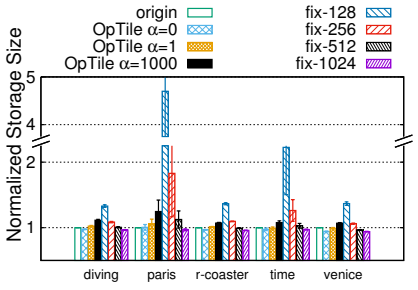
The results are shown in Figure 6. In this figure, we normalize the downloaded bytes against the number of bytes downloaded without tiling. By incorporating the view downloading cost into the integer programming objective and increasing the associated weight α, the downloading volume can be greatly reduced. When $\alpha = 1000$, OpTile always performs the best, resulting in the least downloading volume. Compared to the original scheme without tiling, OpTile can save 62% to 71% downloading volume for 1920 × 960 videos, 64% to 73% downloaded volume for 3840 × 1920 videos. Compared to the best-performing fixed tiling scheme, OpTile can reduce the downloaded volume by 16% to 44% for 1920 × 960 videos, 11% to 34% for 3840 × 1920 videos.

For the *fix-n* schemes, no scheme performs the best for all videos. Small tiles incur high storage overhead, but deliver a small number of pixels per tile, reducing the number of wasted pixels (i.e., pixels that are downloaded in a tile but never rendered), while large tiles result in the opposite behavior. Furthermore, we see very high variations in normalized downloaded volume across two videos. These variations are due to the high variation in storage overhead of fix-sized tiles. OpTile schemes produce more stable results than the *fix-n* schemes.

5.2.2 Naive Prediction. In addition to evaluating the network cost under perfect prediction, we also measure the downloading volume of 360-degree video streaming with inaccurate prediction. In this experiment, we employ a naive prediction algorithm. This algorithm predicts the user's head orientation three seconds after any point in the video. Prediction is performed by assuming the user's head orientation does not change position after three seconds has passed. For a temporal segment with its playback deadline in three seconds, the client first downloads tiles that would cover the naively-predicted viewport. We also assume that the client makes perfect predictions one second before the playback deadline. If the three-second prediction was incorrect, we allow the client to use its perfect one-second prediction to download any missing tiles in the user viewport.

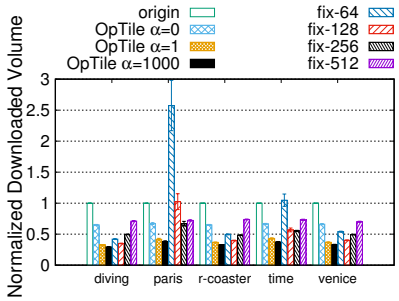


(a) 1920×960

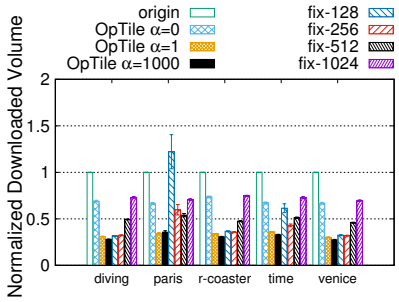


(b) 3840×1920

Figure 5: Normalized storage size with various cutting schemes.

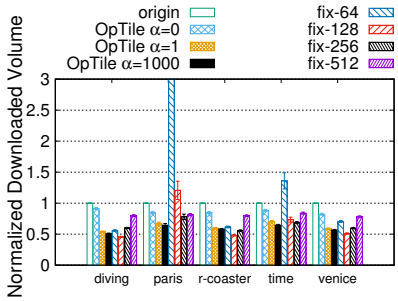


(a) 1920×960

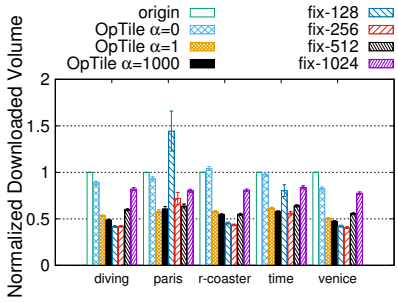


(b) 3840×1920

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



(a) 1920×960



(b) 3840×1920

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

Therefore, the total downloading volume of a temporal segment is the sum of the costs to download the tiles from the three-second prediction and the remaining tiles downloaded from the perfect one-second prediction.

The results are shown in Figure 7. Compared to perfect prediction, the downloading volume of all tiling schemes has increased. When $\alpha = 1000$, our scheme performs the best in two out of five videos: "paris" and "time". In the rest three videos, OpTile's downloading volume is within 25% of the best fixed-tiling scheme. Compared to the original scheme without tiling, OpTile can save 35% to 49% downloaded volume for 1920×960 videos, 39% to 53% downloaded volume for 3840×1920 videos.

Although fixed-tiling schemes may perform better for these three videos, their performances vary significantly. For example, for videos in 1920×960, *fix-128* performs the best for "diving", "r-coaster" and "venice", but performs even worse than the original scheme without tiling for "paris". For videos in 3840×1920, *fix-128* performs the best for "diving", but worse for "paris" and "time" among all fixed-tiling schemes. This performance variation is not unexpected, as fixed tiling schemes do not take video content characteristics or user watching behavior into account when making tiling decisions. On the other hand, when setting $\alpha = 1000$, OpTile always significantly outperforms the original non-tiled projection scheme and performs within 25% of best-performing fixed tiling scheme.

6 CONCLUSION

In this work, we outlined a fundamental problem in 360-degree video streaming: wasted bandwidth associated with unseen portions of the 360-degree view. This bandwidth problem has been addressed by two main approaches. Oriented projections define a direction of improved quality; a user viewport oriented in this direction will experience greater visual quality with lower bandwidth consumption. Tile-based approaches cut the 360-degree view spatially into disjoint rectangular sections. At best, these approaches are able to send only the tiles needed to cover the viewport.

We attempt to advance the state of tile-based approaches by considering how to best tile a projected 360-degree surface to optimize combined use of storage and download bandwidth. We formulate these storage and bandwidth concerns as an ILP. Experimentation by running the ILP on a training set of user head orientations and evaluating the solution's storage and bandwidth costs on a disjoint test set of user head orientations shows that these non-uniform ILP tiling solutions can significantly outperform existing tilings schemes.

7 ACKNOWLEDGEMENT

We appreciate constructive comments from anonymous referees. This work is partially supported by NSF under grants CNS-1524462 and CNS-1618931 and a fund from Adobe Systems.

REFERENCES

[1] 360-degree videos head movements dataset. http://dash.ipv6.enstb.fr/headMovements/.

[2] 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.

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

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

[5] Glpk (gnu linear programming kit). https://www.gnu.org/software/glpk/.

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

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

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

[9] Next-generation video encoding techniques for 360 video and VR. https://code.facebook.com/posts/1126354007399553/next-generation-video-encoding-techniques-for-360-video-and-vr/.

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

[11] C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[12] X. Corbillon, A. Devlic, G. Simon, and J. Chakareski. Viewport-adaptive navigable 360-degree video delivery. *arXiv preprint arXiv:1609.08042*, 2016.

[13] 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.

[14] 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.

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

[16] 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.

[17] M. Hosseini and V. Swaminathan. Adaptive 360 vr video streaming: Divide and conquer! *arXiv preprint arXiv:1609.08729*, 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] O. A. Niamut, E. Thomas, L. D'Acunto, C. Concolato, F. Denoual, and S. Y. Lim. MPEG DASH SRD: spatial relationship description. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 5. ACM, 2016.

[22] D. Salomon. *Transformations and projections in computer graphics*. Springer Science & Business Media, 2007.

[23] G. J. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[24] A. Taghavi Nasrabadi, A. Mahzari, J. D. Beshay, and R. Prakash. Adaptive 360-degree video streaming using layered video coding. In *Virtual Reality (VR), 2017 IEEE*, pages 347–348. IEEE, 2017.

[25] 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.

[26] M. Yu, H. Lakshman, and B. Girod. Content adaptive representations of omnidirectional videos for cinematic virtual reality. In *Proceedings of the 3rd International Workshop on Immersive Media Experiences*, pages 1–6. ACM, 2015.

[27] 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 the 2016 ACM on Multimedia Conference*, pages 601–605. ACM, 2016.

[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.