L3BOU: Low Latency, Low Bandwidth, Optimized Super-Resolution Backhaul for 360-Degree Video Streaming

Ayush Sarkar*, John Murray†, Mallesham Dasari‡, Michael Zink†, Klara Nahrstedt*
*University of Illinois Urbana-Champaign, †University of Massachusetts Amherst, ‡Stony Brook University
*{ayushs2, klara}@illinois.edu †jomurray@umass.edu, zink@ecs.umass.edu ‡mdasari@cs.stonybrook.edu

***Abstract*—In recent years, streamed 360° videos have gained popularity within Virtual Reality (VR) and Augmented Reality (AR) applications. However, they are of much higher resolutions than 2D videos, causing greater bandwidth consumption when streamed. This increased bandwidth utilization puts tremendous strain on the network capacity of the cloud providers streaming these videos. In this paper, we introduce L3BOU, a novel, three-tier distributed software framework that reduces cloud-edge bandwidth in the backhaul network and lowers average end-to-end latency for 360° video streaming applications. The L3BOU framework achieves low bandwidth and low latency by leveraging edge-based, optimized upscaling techniques. L3BOU accomplishes this by utilizing down-scaled MPEG-DASH-encoded 360° video data, known as Ultra Low Resolution (ULR) data, that the L3BOU edge applies distributed super-resolution (SR) techniques on, providing a high quality video to the client. L3BOU is able to reduce the cloud-edge backhaul bandwidth by up to a factor of 24, and the optimized super-resolution multi-processing of ULR data provides a 10-fold latency decrease in super resolution upscaling at the edge.**

***Index Terms*—360° video, video streaming, super-resolution, edge computing, bandwidth, latency**

I. Introduction

360° videos achieve unique visual immersions temporally and spatially, allowing users to view panoramic content by seamlessly altering the locations of their viewports through Head-Mounted Displays (HMDs) and players amenable to 360° media. 360° video content is currently on the rise with the proliferation of devices that facilitate Virtual Reality (VR) and Augmented Reality (AR) video traffic, which is expected to reach 4.02 exabytes per month by 2022 [16]. Streaming panoramic videos does not come without added challenges; even the most lax Quality of Experience (QoE) benchmarks pertinent to 360° use cases necessitate demanding data requirements. 360° videos require significantly higher *network bandwidth and bitrates* than those of conventional videos to achieve similar perceived quality levels. Furthermore, the proximities of HMD displays that enable 360° content viewing to users' faces dictate an even greater emphasis on *low latency* requirements. Similar to 2D videos, 360° videos can be streamed via Dynamic Adaptive Streaming over HTTP (DASH) [7], with each video being segmented temporally on the server side. Since significant portions of 360° frames are not viewed due to the user's limited viewing angle, the intensive network bandwidth requirement can be curbed through viewport adaptive streaming approaches (e.g., [6], [7]) that utilize viewport prediction algorithms to only download content restricted to the user's viewport. Each temporal segment is partitioned spatially into tiles to enable viewport adaptive streaming.

Mass delivery of 360° content presents tremendous bandwidth and latency challenges for cloud providers. Typical streaming solutions rely on cache servers located at network edges. These edge caching systems aim to alleviate backhaul network congestion while achieving low latency by providing data storage geographically closer to clients. The network load will further increase in the future, and thus cloud providers need to find innovative strategies to lessen core network bandwidth consumption and facilitate the delivery of low latency content, tackling the additional video streaming growth resultant from the influx of new users. However, the solutions for conventional videos face difficulties when applied to 360° streaming scenarios. 360° video frames are of much higher resolutions and only a small portion of the available scene (the viewport) is delivered to the client at a time. These challenges for 360° content streaming are further exacerbated by DASH-based video caching, where multiple quality versions (expressed in bitrates) of each segment need to be stored to enable Adaptive Bit Rate (ABR) streaming with different quality levels. Existing solutions propose that clients utilize their own compute capability to help reduce network bandwidth [13] [18]. However, these solutions do not perform well on client devices with lower compute capacities. A content-aware approach is thus needed in order to explicitly consider the content semantics of each video for efficient data transfer, decreasing network bandwidth while also supporting resource-constrained client devices.

We approach this problem by exploring the benefits of a classical image problem, super resolution (SR), for lowering backhaul bandwidth and minimizing playback latency. SR techniques look to generate or recover a high resolution (HR) version of low resolution (LR) inputs, typically through deep-learning based frameworks. NAS [25] was the first to leverage super resolution for streaming scenarios involving conventional videos by trading off network bandwidth for client-side compute, fetching downscaled, low resolution content over the network and reconstructing the high resolution content at the client. PARSEC [4] expanded on this idea for 360° content, handling challenges stemming from the large sizes of 360° videos by training smaller SR micro-models for each segment that would focus only on upscaling tiles rather than entire frames to reduce computational overhead. However, these approaches operate under the assumption that the network is a simplistic client-server architecture, ignoring popular DASH streaming scenarios that involve edge nodes with greater compute. By restricting the problem scope, previous frameworks incur greater energy overheads on the client-side due to GPU usage for SR upscaling, leading to increased heat dissipation. PARSEC, for example, only focuses on using SR to upscale missed tiles to correct for viewport prediction inaccuracies; this does not fully take advantage of all of the possible benefits of SR with regards to saving network bandwidth.

In this paper, we present L3BOU, a novel, distributed architecture for 360° video streaming. In contrast with existing approaches, the super-resolution process is distributed over the server, edge, and client. The inclusion of edge compute in the distribution process allows for high-quality streaming of 360° videos to resource constrained clients. L3BOU reorients the problem towards alleviating backhaul network congestion while also minimizing client-side computation, yielding low bandwidth for backhaul cloud-edge networks and low latency for resource-constrained clients. L3BOU leverages the increased compute capacity of edge nodes to request downscaled, Ultra Low Resolution (ULR) tiled segments from the cloud and run upscaling SR algorithms on all edge-present ULR tiles, relying on pre-trained SR micro-models [4]. To achieve low latency and fulfill video quality goals despite the increased computational complexity, we further utilize (a) a viewport prediction algorithm based on historical viewing patterns and represented as a Navigation Graph (NG) construct [17], (b) an NG-based *prefetching mechanism*, and (c) multi-processing of SR tasks across edge server processors to achieve parallelism. The main contributions of this paper are as follows:

1) We develop L3BOU, a three-tier distributed framework where the cloud serves ULR and full quality tiles to the edge. The edge node serves as a cache layer to store trained SR micro-models, performs viewport prediction, and super-resolves ULR tiles. This approach achieves significant cloud-edge network bandwidth optimization and ensures that the resource-constrained client is not involved in the SR computations but still receives high-quality tiled segments within its playback time, avoiding re-buffering events.
2) We perform multiprocessing of SR deep learning inference tasks *during playback* at the edge, further speeding up the inference rate; this happens while leveraging the powerful GPUs present on edge nodes.
3) We mask latency overheads through (a) a unique usage of the Cross-User Navigation Graph viewport prediction algorithm [17], (b) the prefetching of heavily compressed, downscaled ULR tile segments, and (c) taking into account encoding, decoding and SR inference timings.
4) Our evaluation results show that the back-haul bandwidth between cloud and edge can be reduced by upto a factor of 24, and that the upscaling time of ULR tiles at the edge can be reduced by factor of 10 while the resulting QoE is comparable to the one achieved by existing approaches that do not incorporate the presented optimizations at the edge.

The paper is organized as follows: In Section II we discuss models, assumptions and other related work utilized in our L3BOU framework, giving brief introductions to DASH, the Navigation Graph construct and its associated viewport prediction, and content-aware super-resolution approaches. Section III presents the offline phase with super-resolution training algorithms to facilitate the creation of ULR tile segments and micro-models, the online phase of L3BOU with its individual service components across the three-tier architecture to deliver high quality desired tiles to clients, and the performance optimization approaches involved. Extensive performance evaluation results are presented in Section IV. Section V summarizes L3BOU's key architectural points and results.

II. Models, Assumptions, and Related Work

A. *360° Video Streaming Model*

We use the DASH standard for 360° video streaming. DASH videos are segmented temporally at the server-side, with each of these temporal segments representing a few seconds of media playback. These segments are then encoded at different quality levels/bitrates, and a manifest file that consists of the necessary metadata and ordering of these segments is generated. The client typically utilizes an adaptive bitrate (ABR) algorithm that dynamically adjusts the bandwidth transmitted between server and client at run-time in response to network throughput variations, using the manifest file to automatically request segments of the highest possible bitrates without incurring re-buffering events.

Viewing clients of 360° videos cannot view all portions of the panoramic scene simultaneously. Hence 360° streaming systems conduct viewport predictions to deliver desired viewports to clients, decreasing the network bandwidth demand. Each temporal segment of the 360° video is partitioned spatially into tiles, and only tiles of the user's current viewport are transmitted to the client device. The client's ABR algorithm determines the bitrate for each tile.

System Assumption: We assume that our 360° videos follow an equirectangular projection [27] and are encoded for DASH streaming. We also assume that DASH segments are of a one second duration.

B. *Navigation Graph*

The Navigation Graph [17] concept models clients' viewing patterns of a 360° video. The Navigation Graph model is defined as follows: a set s is the union of all visible tiles in a particular segment of the 360° video such that $s \in \mathbb{S}$, where $\mathbb{S}$ is the set of all possible combinations of tiles. A "view" is a tuple delineating the union of all visible tiles of a viewport within a particular segment. Hence, "view" $v = (l, s)$ represents a set of visible tiles s in a segment with index l. The Navigation Graph $G = (V, E)$ is then a directed graph that models the union of all clients' past viewing behaviors per 360° video. The vertices of the graph are defined as

$$V = \{v | v = (l, s), l \in \{1, 2, ..., L\} \text{ and } s \in S\}, \quad (1)$$

where L is the total number of segments in the video. An edge in G is created between vertices when at least one view transition occurs. E can be expressed as:

$$E = \{(v_i, v_j) | v_{i,j} \in V, w(v_i, v_j) = p(v_j | v_i), i, j \in 1, .., N\}, \quad (2)$$

where $w(v_i, v_j)$ is a weight function, N is the total number of vertices visited at least once, and $p(v_j|v_i)$ is the probability of a viewer transitioning to v_j from v_i. We also denote matrix $\hat{E} = \mathbb{R}^{N \times N}$, which stores the transition probabilities between all vertices in G. A Navigation Graph G is constructed for each 360° video, and it continues to gain viewing pattern information as new clients watch the video. A new vertex in G is created when a new view v_n is encountered, and the transition probabilities $p(v_n|v_c)$ (with v_c being the current view) are updated on the server based on the influx of viewing data from new clients.

System Assumption: In L3BOU, we assume that a Navigation Graph is created and available on the cloud server side for each 360° video, based on prior viewing patterns of each 360° video, executing the Cross-User Navigation Graph View Prediction (CU) approach. In the CU approach, the graph is updated by the view transitions from all prior viewers. Each transition probability in matrix $\hat{E}$ is updated according to

$$p(v_c|v_p) = \frac{\text{\# of clients moving their view from } v_p \text{ to } v_c}{\text{number of clients visiting } v_p}. \quad (3)$$

The probability that a client viewer will change from view v_c to v_n is provided by the column vector of matrix $\hat{E}$, denoted as b_1, which contains elements $p(v_n|v_c)$ for $1 \le n \le N$. The k^{th} transition vector, b_k can be expressed as

$$b_k = \hat{E}^{(k-1)} b_1 \quad (4)$$

to represent the probabilities from the current vertex to vertices in a segment with index $l + k$. The probability of needing a tile t can then be given as

$$p_{t,k} = \sum_{\forall n, t \in v_n} b_k^n. \quad (5)$$

L3BOU then prefetches the Navigation Graph from the cloud, and utilizes the Navigation Graph on the edge node.

C. Data Distribution Model

We assume a three-tier distributed architecture, consisting of a *cloud server*, an *edge server* with available computing resources such as GPUs or TPUs and a *resource-constrained client device* (e.g., Head-Mounted Device or Smartphone). The client device runs a DASH-based 360° media player, with the edge serving DASH segments. 360° videos are segmented, tiled and processed based on the MPEG-DASH SRD standard and transmitted between the cloud server and edge over HTTP.

D. Content-Aware Super Resolution

In addition to traditional compression of 360° videos (e.g., using HEVC), we consider using Super Resolution [11], [12], [23] (SR) for further content-aware compression. Using SR, the 360° video is compressed significantly, and an appropriately trained neural SR model can generate a high resolution video from the compressed video. Recent developments in computer vision literature have seen tremendous success in super-resolving low quality videos using convolutional neural networks (CNNs) [12], [25].

Note that video applications already encode/compress videos to reduce bandwidth demand using compression methods such as HEVC. However, traditional compression methods use fixed configuration parameters such as the search space of redundant pixels in a given image. They also use excessive quantization under low encoding rates [20]. SR, however, has complete search space of an image, automatically determines the high level features of image objects, stores the latent information in model weights, and then enhances particular regions to high resolution from the low resolution. The combination of both SR model weights and the pixel level information from the low resolution image creates a high quality image [19]. Recent work in this space already demonstrated the superiority of SR along with traditional compression mechanisms [4], [25].

However, the key challenges in leveraging SR for DASH-based 360° video streaming are large model sizes and inference times. To address these challenges, we use the concepts of *micro-models* and *Ultra Low Resolution (ULR) tiles* introduced by PARSEC [4]. To perform a super-resolution task for 360° streaming scenarios, the video is initially downscaled from its original high resolution to a lower resolution. The downscaled video is then further segmented, tiled, and packetized for DASH delivery - we refer to each of these heavily compressed, encoded tiled segments as a ULR tile. During the streaming process, these ULR tiles are decoded and sent to specific SR micro-models, which are small neural network architectures that are specifically over-fitted to the target content beforehand in order to reconstruct the high resolution content from the decoded low resolution input before re-encoding for DASH re-delivery. Each micro-model is trained to upscale frames only corresponding to a particular segment of the video. Since each micro-model is trained for a segment (e.g., 30 video frames for 1 second segment), the model is extremely lightweight, resulting in faster inference as well as smaller model size. Another key benefit of this approach is that we do not need large-scale datasets for the SR model, as the model is over-fitted with just the segment data.

III. L3BOU System Architecture

The L3BOU three-tier distributed software architecture and framework extends super-resolution 360° video streaming approaches to achieve low bandwidth and low latency for 360°

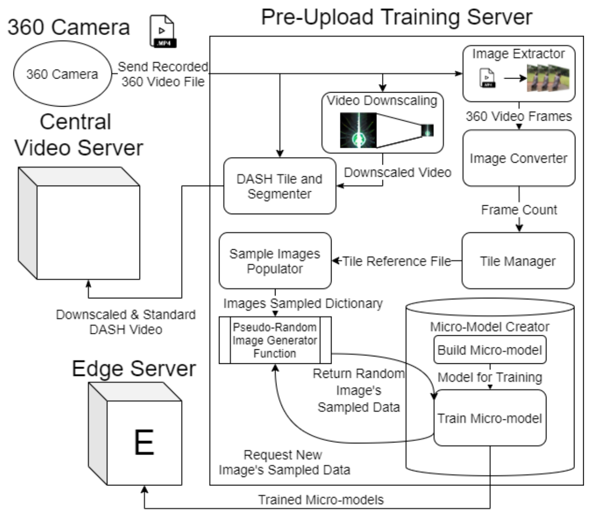

Fig. 1. L3BOU Offline Phase Architecture

video traffic over backhaul cloud-edge networks. The L3BOU framework consists of two phases: an *offline phase* and an *online phase*. During the offline phase, shown in Fig. 1, the pre-upload training server performs DASH segmentation, tiling of 360° videos, downscaling of tiles towards ULR tiles, and trains super-resolution micro-models for ULR tile upscaling. The DASH-based ULR tiled segments are then placed on the cloud and the trained micro-models are placed on the edge. A detailed discussion of L3BOU's offline phase is presented in section III.B. L3BOU's online phase is shown in Fig. 2. During this phase, the cloud server relies on a DASH-based 360° video representation consisting of downscaled segmented tiles and on the Navigation Graph to provide context assistance for viewport prediction. The online phase of L3BOU also responds to DASH requests from the edge server. The L3BOU edge server then performs upscaling tasks via a parallelized super-resolution approach and communicates with a viewing client to deliver a high quality viewport. We present detailed algorithms of the L3BOU online phase in Section III.C.

A. L3BOU Offline Phase

The L3BOU offline 360° video processing phase is handled by a pre-upload training server. In a real world scenario, such processing can occur at the cloud data center or at the edge, but for simplicity, we illustrate this framework by assuming the existence of a training server managed by cloud providers. The core component of the pre-upload training server is the *Micro-Model Creator* module, which trains and prepares micro-models for each ULR tiled segment so that the edge servers can then upscale each tile, i.e., apply super-resolution algorithms, bringing each video segment's tiles to a high video quality.

Super Resolution Micro-model Architecture: Our SR DNN (Deep Neural Network) architecture is similar to that of PARSEC and NAS. The SR DNN's number of layers depends on the length of the video segment and the desired quality of the generated video. We use a deep convolutional neural network (CNN) to capture high level features in the video segment. Each convolutional layer is followed by a LeakyRelu activation function [14] and Batch normalization for faster learning [8]. The neural network first extracts the high level features from low-level pixels and uses a non-linear mapping function to learn the original missing content details. Finally, the network uses a deconvolution layer to map the high resolution directly from the low resolution without image interpolation. The Adam optimizer [9] is used for training with a learning rate of 0.0002. We adopt smaller filter sizes (3×3) to minimize the model parameters as much as possible and use more mapping layers at the expense of more computation.

The pre-upload training server processes the 360° videos as follows: it first divides the video into 1 sec long segments. Each segment is divided spatially into tiles. Each tile (128×120 pixels, with the tiling scheme selected to avoid excluding any pixel data) is downscaled to a lower resolution through *ffmpeg* [1] (such as 24×24 pixels) and then further encoded to produce ULR tiles. For training, these ULR tiles are decoded into sets of frames. Each low resolution frame is attached to its respective ground truth high resolution frame to form a training pair, and these training pairs are input into the micro-model specifically responsible for the segment of the video that the original ULR tile is from. The training of each micro-model takes less than 10 minutes (on our platform as described in Section IV) because we train the model with very little data (i.e., a segment). Each model is trained for one video segment (with 30 fps). Therefore, the total number of ULR and ground-truth high resolution images is 30 for each model (assuming 30 fps video). While training, we use the MSE metric as the loss function to directly optimize for PSNR. The Adam optimizer aims to optimize the PSNR for every frame of the video segment. We manually fine-tune the number of layers and filter size to achieve a desired median quality (about 30 dB PSNR) [21] when mapped from ULR tiles to high resolution, which is considered the minimum quality necessary for a good quality of experience. To facilitate ABR streaming at different quality levels, we construct a micromodel for each segment-bitrate combination, meaning that different micromodels attached to the same segment index are responsible for upscaling at different quality levels.

Offline Phase Workflow: Initially, the 360° videos are recorded through omnidirectional cameras and uploaded to the training server as *mp4* files in equirectangular format. The video files are then sent to the *Frame Extractor*, which takes each frame in the video file, saves it to a local hard drive, and partitions the input frames by segment index. The frames corresponding to each segment are then sent to the *Image Converter*, which spatially tiles each frame and encodes the frame data into a lower level format, capturing the RGB values of each pixel. The encoded frame representations are saved to a file on the local system and compiled tile metadata is passed to the *Tile Manager*, which generates a local file used to reference the encoded frames.

This file is subsequently referenced to populate a dictionary

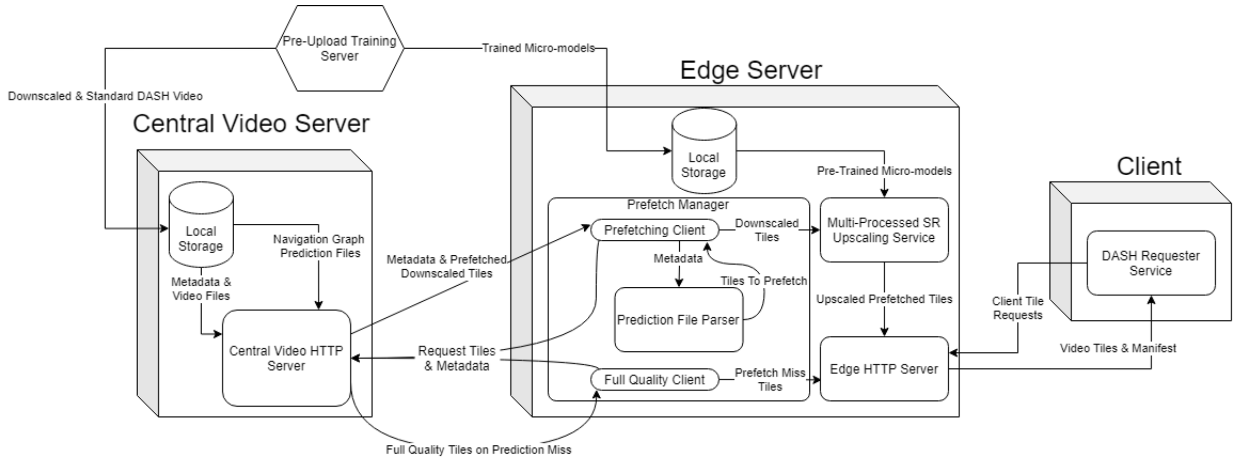

Fig. 2. L3BOU Online Phase Architecture

with the low level frame data. The dictionary is then utilized by the generator, which returns a set of pseudo-random image data obtained from the dictionary. A SR micro-model for each segment-bitrate combination is constructed, and each micro-model uses the generator to obtain image data for training. Once each micro-model is fully trained, it is pushed to the edge server.

Optimization - Edge Node Cache Utilization: The function of the edge cache in the L3BOU framework fundamentally deviates from that of a typical edge-native, static CDN (Content Delivery Network).The edge node is not used as a simple cache layer for all of the media content from the cloud. Instead, the edge node only caches the requisite micro-models after offline training. This is a network bandwidth optimization, relieving the cloud server from having to send the corresponding micro-models for each segment during video playback. Once all micro-models are trained offline, they need to be offloaded to the edge server one time. Once they are deployed, the edge server is now aware of the content and only needs to retrieve the necessary ULR tiles from the cloud during playback for all subsequent streaming scenarios facilitated by L3BOU.

B. L3BOU Online Phase

Once the micro-models for each segment/tile are trained offline and deployed to the cache storage of the edge server, L3BOU is amenable to SR-enhanced video streaming and playback. We assume that prior to the L3BOU's online phase, 360° video segments/tiles are present in two forms at the cloud server: (a) full quality segment/tiles and (b) downscaled segments/tiles to the ULR representations, and both full quality and the ULR segment tiles of the 360° videos are present in its local storage. In addition, a DASH manifest file (MPD) is generated for each 360° video. Furthermore, the cloud server stores the Navigation Graph (as a JSON file) for each 360° video, having the information about prior viewers' viewing patterns of 360° videos, i.e., the Navigation Graph provides the percent chance (probability) of each ULR tile being viewed for a specific segment. These files, including segments with ULR tiles to be viewed with high probability, are then served through a threaded HTTP server from the cloud to the edge server during playback.

Online Phase Workflow: When the client requests a video through DASH, the request is forwarded via edge server to the cloud, which initially returns the manifest file and the video's corresponding Navigation Graph JSON to the edge. The Navigation Graph matrix, discussed in Section II.B, is then utilized by the HTTP Prefetching client on the edge to commence prefetching ULR tiles to upscale, while the manifest is forwarded to the client for use. The client proceeds to request video tiles corresponding to the current viewport as the edge server upscales tiles that are anticipated to be delivered. When the edge server receives a request from the client, two possible cases can occur:

Case 1: The client requests a tile accurately predicted by the Navigation Graph. In this case, the ULR tile was prefetched, decoded, upscaled by the micro-model corresponding to the current segment index, and re-encoded for delivery to the client. This leads to the upscaled, full-quality tile being delivered at or ahead of its playout deadline.

Case 2: When a tile is requested by the client that was not prefetched and upscaled, indicating a tile miss, the edge server forwards the request to the cloud server. The cloud server then delivers the full-quality tile to the client without any SR processing involved to minimize latency overhead.

The Online Phase consists of the following five software components:

1) Central Video HTTP Server (Cloud Server): The central video server caches ULR tiles and full quality tiles in its local storage. The central video HTTP server references these tiles to serve them to the edge server upon receiving an HTTP request.

2) Prefetch Manager (Edge Server): The *Prefetch Manager* first utilizes a *Prefetching Client* to request the metadata of the video, which consists of the Navigation Graph file and the manifest file. The Navigation Graph is passed to the *Prediction*

File Parser, which decodes the graph and extrapolates which ULR tiles to prefetch during that prefetching cycle. As stated before, the Navigation Graph prediction file consists of a set of tile probabilities for each view v. However, each view is the union of *all* visible tiles within a segment. Since the Navigation Graph learns the historical viewing patterns of many users in order to learn tile probabilities, in most cases there are a myriad of visible tiles with a very low, but nonzero probability of occurrence. Therefore, we define the set of ULR tiles s_k to be prefetched for a segment as

$$s_k = \{t \in s | p_{t,k} \geq \alpha\}, \tag{6}$$

pruning all tiles t with a probability below a cutoff threshold of α. The *Prediction File Parser* then lets the *Prefetching Client* know which tiles to prefetch from the cloud. The *Prefetching Client*, in response, proceeds to request the ULR tiles in s_k, and once they are received, they are then forwarded to the *Upscaling Service*. In cases where a tile miss occurs, the *Full Quality Client* requests the respective full quality tile so that it can be forwarded to the client.

3) Upscaling Service: The Upscaling Service on the edge server executes the SR techniques and leverages the increased number of CPU cores in tandem with the powerful GPUs present on the edge server for task parallelism during video playback. After all ULR tiles in s_k are decoded serially, each decoded low resolution tiled segment is viewed as a series of frames.

Each frame is indexed from 1 to N, with N being the total number of frames per decoded segment (e.g., if each viewport includes 6 ULR tiles in the spatial view, and the overall segment is 1 second with $N = 30$, then we will have 6 tile-sized videos of 1 second length for a total of 180 tile-sized frames).

Each segment is 1 second long for a 30 fps video, so our L3BOU software architecture assumes $N = 30$ for each prefetching cycle. Multiprocessing is then employed to distribute ULR frames between GPU-accelerated processes, such that each worker performs SR deep learning inference tasks only on frames with indices within an allocated frame partition.

4) Edge HTTP Server: The *Edge HTTP Server* receives requests from the client and first checks if the requested ULR tiles have been super-resolved in a previous or the current prefetching cycle and are available to be distributed. If the super-resolved tile is ready, the *Edge HTTP Server* forwards it to the client. If the tile has not been super-resolved and is unavailable, then a tile miss has occurred. A tile miss can occur from two situations: either the Navigation Graph failed to accurately predict the tile, leading to a viewport prediction inaccuracy, or the tile was accurately predicted but the available compute is not sufficient to perform super resolution on the prefetched tile fast enough before playback time. In both cases, the *Full Quality Client* will request the full quality tile needed from the *Central Video HTTP Server* and return the result back to the client.

5) Client DASH Requester Service: The *Client DASH Requester Service* initially requests the manifest file of the video from the edge server. Upon reception of the manifest, the client proceeds to request DASH tiled segments based on the user's viewport position.

C. Achieving Computational Feasibility

In this section, we demonstrate how optimized super-resolution during video playback can facilitate the functionality of L3BOU.

We denote a function $q_k : s_k \rightarrow Q$ to illustrate the rate selection algorithm for ABR streaming, mapping ULR tiles $t \in s_k$ to selected quality levels Q (Q is expressed in bitrates). Instead of parallelizing the super-resolution inferences, we first assume that all prefetched ULR tiles $t \in s_k$ are super-resolved in a serial manner. Thus, we can define the processing time for all ULR tiles on the edge during a single prefetching cycle, T_s, as

$$T_s = \sum\nolimits_{t \in s_k} \left(SR(t, q_k(t)) + \beta_t + \gamma_t\right), \tag{7}$$

where β_t and γ_t represent the decoding and re-encoding times for tile t, respectively. $SR(t, q_k(t))$ represents the time taken to upscale ULR tile t to a selected quality level of $q_k(t)$. The *playback constraint* is a constraint that illustrates that the fetching and super-resolution of all tiles in s_k must complete before the designated playout time of the segment. This is modeled by

$$|s_k|\lambda + T_s + \sum\nolimits_{t \in s_k} D(t, q_k(t)) + \delta \leq \xi, \tag{8}$$

where λ represents the time it takes to fetch an individual ULR tile from the cloud server, $D(t, q_k(t))$ represents the time taken to deliver a tile t upscaled to its selected quality level from the edge to the client device, δ represents the computational time taken by the client for decoding and stitching once the upscaled tile has been received, and ξ represents the time until playback. This constraint represents the scenario where all ULR tiles, prefetched based on the Navigation Graph predictions, are super-resolved before the client requests them - if this constraint is not satisfied, then even if the tiles are accurately predicted by the Navigation Graph, they will still be considered as tile misses.

With the previous formulation of T_s in which all super-resolution tasks are conducted sequentially, the playback constraint is still not satisfied and the solution still remains infeasible despite the artificial increase of ξ through prefetching. Despite the increased compute capacity of the edge server, the time taken for each SR task on an entire segment takes multiple seconds, leading to unacceptable latency overheads that cancel out the buffer time provided by prefetching. The benefits provided by prefetching are not significant enough *by themselves* to outweigh the computational overhead from T_s. Therefore, L3BOU relies on task parallelism (multiprocessing) during video playback to reduce T_s, which is incorporated within the *Upscaling Service*. We denote the processing time for all ULR tiles on the edge during a single prefetching cycle with parallel processing as T_p. We find that with enough GPU-accelerated processes running, $T_p << T_s$, and thus, we see that by substituting T_p for T_s in (8), the playback constraint is satisfied with the artificial increase of ξ through prefetching. Through the reduction of T_s to T_p with SR task parallelism along with the latency cushion provided by prefetching, L3BOU achieves two fundamental latency relaxations regarding the backhaul links: we can afford to use a significantly lower bandwidth link and/or handle increased delays from the link between the cloud and edge.

IV. Performance Evaluation

Our testing was performed with two different systems representing the edge server with different levels of computational power. The first system, which will be referred to as *Testbed 1*, consisted of an enthusiast grade desktop with a twenty core CPU and an NVIDIA 2080 super for Tensorflow [5] acceleration. The second system, which will be referred to as *Testbed 2*, was a n1-standard-32 Google cloud instance with 32 vCPUs and dual NVIDIA T4 Teslas for Tensorflow acceleration. Both of these systems were used for testing the Online and Offline phases as described in Sect. III at separate times for measurement of training metrics and SR timings. When performing certain tests involving the video server and/or client, a laptop was used to act as the cloud server or a simulated DASH client. The cloud server was facilitated by running a Python-based multi-threaded HTTP server. However, the simulated DASH client fetched video files using the standard Python HTTP client and a section of specialized code that decided which and how many tiles to prefetch. For our evaluation, we used a publicly available dataset [24], containing 8 videos in equirectangular format, each with a resolution of 2560x1440. This decision was made specifically because of its inclusion of previous user head-traces, where the viewing pattern data for all users was already available for the use of a Navigation Graph.

During our testing, we measured the overall amount of data transferred between our L3BOU cloud and edge servers at various prefetch success rates. These prefetch success rates were based on the total tiles that were successfully predicted, upscaled and sent to the client while also taking into account any prediction misses involving the streaming of full quality tiles with a resolution of 128x120 pixels. These tiles were created using the Kvazaar [22] encoder paired with GPAC's MP4Box [10] for DASHing. Changing the model settings of our micromodels was also considered, ensuring that our output video's PSNR value was acceptable at 30 dB or above. Finally, to further decrease the system latency resultant from the super resolution tasks, multi-processing was employed. Our testing included running the super resolution tasks with a varying number of processes available and modifying the number of convolutional layers for our micromodels.

A. *Impact of Changes in Pre-trained Model*

Table 1

Edge Testbed 1 Model Results

Model Layer Count	Epoch Count	24x24 Tile Model Training Time (mm:ss)*	24x24 Tile Model PSNR (dB)*	48x48 Tile Model Model Training Time (mm:ss)*	48x48 Tile Model Training PSNR(dB)*
1	6000	06:20	29.32	05:58	30.47
5	6000	07:49	30.01	07:48	30.89
12	6000	11:28	30.37	11:29	31.71

* Results displayed are the average of 5 test runs.

Table 2

Edge Testbed 2 Model Results

Model Layer Count	Epoch Count	24x24 Tile Model Training Time (mm:ss)*	24x24 Tile Model PSNR (dB)*	48x48 Tile Model Model Training Time (mm:ss)*	48x48 Tile Model Training PSNR(dB)*
1	6000	07:00	29.36	06:55	30.11
5	6000	12:21	30.33	12:31	31.14
12	6000	30:06	30.69	29:30	31.26

* Results displayed are the average of 5 test runs.

L3BOU's SR models were created using Keras [3] and Tensorflow 2.2.0, and changes made to creation and training resulted in the following system inputs being taken into consideration: *convolutional layer count, downscaled tile size, number of training epochs*, and the *batch size*. The output metrics were the overall model training time and the PSNR test output values.

Starting with the *number of epochs* to train for, we began our testing scenarios by evaluating performance with a low number of only one hundred epochs. However, we found that this number of epochs returned inconsistent results. Therefore, we continuously increased this count by one hundred and ran new rounds of model training until our tests showed little increase in performance. Stable performance occurred at around six thousand epochs. We attempted even higher epoch counts (between twenty-thousand and one-hundred thousand epochs), but the improvements in micromodel performance were negligible. Training the models with a significantly higher number of epochs also took exponentially longer than the training time for the six thousand epoch models. The overall model training timing performance analysis is reflected in Table 1 and Table 2.

The *convolutional layer count* experiments showed that, while one convolutional layer was feasible, models required a higher quality Ultra-Low-Resolution (ULR) tile for the PSNR result to remain above the acceptable value of 30 dB. As a result, the model, using one convolutional layer, only functioned with tile sizes of 48x48 pixels and above. Lower resolutions resulted in an unacceptable PSNR as seen in Table 1 and Table 2. Further investigation involved increasing the number of convolutional layers and testing with smaller ULR tile sizes until an acceptable PSNR of 30 dB could be achieved. This testing resulted in finding that five convolutional layers gave acceptable PSNR results for the 24x24 tiles. This was the lowest successful tile size we were able to achieve with our defined micromodel range. A test was also conducted at twelve convolutional layers. A twelve layer model represented the highest threshold of the number of layers we were willing to consider as an acceptable model. This was our limit because after exceeding this threshold, the model files became very large with sizes in tens of megabytes, sizes no longer feasible for our system. In addition, the twelve layer model trained for six thousand epochs failed to upscale the 12x12 and 6x6 tiles in an acceptable fashion, with a sub-30 dB PSNR. While we still included these results, the altered tile size surprisingly had little effect on the model training time and only affected the overall PSNR rating shown in the bottom rows of Table 1 and Table 2. This issue may have been solved with extra training time, however our model training time already increased as more convolutional layers were added. Hence, it would have been impractical to train all of the models required to work with a 12x12 or 6x6 tile size.

Finally, our *batch training size* was set to one third of the count of our frames to upscale. This was achieved experimentally by starting at one and incrementally increasing the count to see the batch size's effect on the PSNR and the training time. While not as drastic as increasing convolutional layers, increasing the sample size increased the training time, and after a certain point, the sample size became detrimental to the PNSR value. Hence, we chose our best performing result at one third the frame count (in our case a batch size of ten). As a result of our experimentations, we chose to focus on models with one and five convolutions, since they returned the best results while keeping a smaller stored model size due to their low number of convolution layers.

B. Multi-Process Upscaling Analysis

To decrease the overall system latency, we chose to multi-process the upscaling of our tiles at the L3BOU edge server. Our testing system consisted of a *worker* program that ran the upscaling tasks and a *manager program*, based on the Python multiprocessing library, that handled all of our processes. The processes started as part of a multiprocessing pool using the `map` function. There was an initial run to gauge the speed of a single process on the system, where all upscaling was done sequentially. However, the testing process then differed in implementation between Testbed 1 and Testbed 2.

In Testbed 1, we were limited to a single GPU with eight gigabytes of memory. As a result, the number of parallel instances we could run was limited. The testing first partitioned the upscaling tasks to every process evenly. The running processes then started simultaneously using the map function and the results were gathered through a console output. Testbed 1 was evaluated using the following numbers of processes in parallel across a single GPU: *two, three, five, and six*. The next possible even split of ten processes would not run on Testbed 1 due to previously stated GPU memory size limitations.

Testbed 2 had a more powerful edge server. With two NVIDIA Tesla T4s, the system had 32 gigabytes of video memory to utilize, and it had double the physical compute power when tasks were shared between GPUs. The testbed as a result was also able to run further tests with ten and fifteen parallel processes.

Fig. 3 shows the results of the single convolutional layer testing on both systems with as little as five SR processes finishing in less than one second (the overall latency threshold). However, this does not take into account the overall utilization

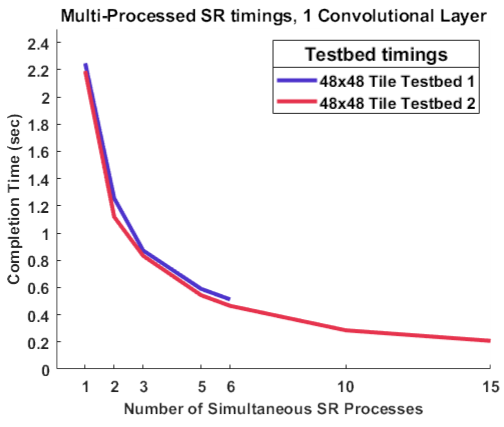

Fig. 3. Single Convolutional Layer SR Graph Comparing Testbeds 1 and 2, Testbed 1 limited to 6 processes due to GPU memory constraints

of the edge server, and as such the 200 ms latency for video processing (breaking segments down to individual frames and reassembling after computation) with 15 running SR processes is much more favorable when such edge compute resources are available. The models in this system setup are also smaller, saving space on the edge - however, the downside to this form of our model is that it requires the use of ULR tiles of 48x48 pixels or 96x96 pixels. This means that more data must be sent over the backhaul network between the L3BOU cloud video server and its edge server.

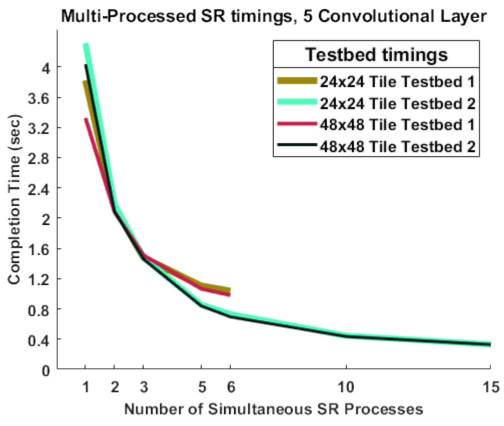

Fig. 4. Five Convolutional Layer SR Graph Comparing Testbeds 1 and 2, Testbed 1 limited to 6 processes due to GPU memory constraints

Fig. 4 displays the further decrease in backhaul network utilization through our use of the five convolutional layer model. This increase in convolutional layers resulted in an improved PSNR rating for tiles with sizes one step lower, indicating that instead of having to use tiles of size 48x48 pixels, our system could use tiles of size 24x24 pixels. This reduces the overall data sent when streaming a video while maintaining the user's QoE. Figures 3 and 4 also demonstrate a 10 times decrease in completion time as shown between 1 and 15 processes.

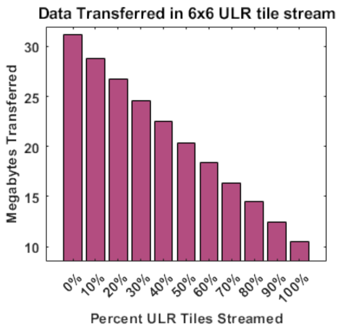

Fig. 5. 6x6 Bandwidth Results

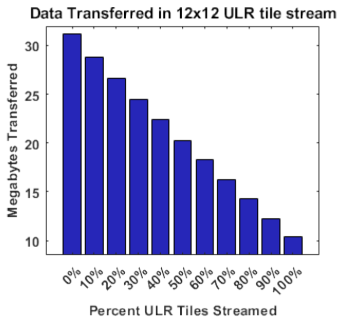

Fig. 6. 12x12 Bandwidth Results

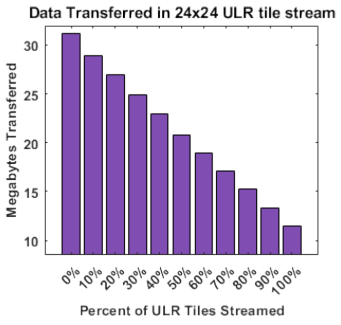

Fig. 7. 24x24 Bandwidth Results

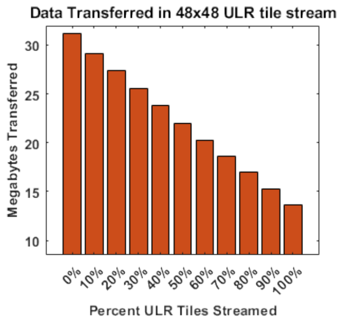

Fig. 8. 48x48 Bandwidth Results

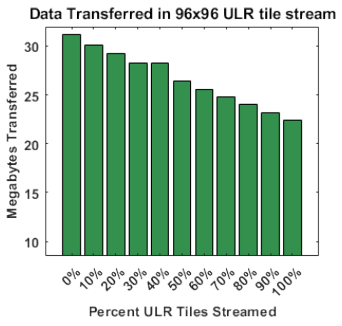

Fig. 9. 96x96 Bandwidth Results

C. Cloud to Edge Bandwidth Savings

Our bandwidth testing consisted of creating five sets of downscaled and tiled DASH streams; each set with a different tile size. These DASH streams were requested by a simulated client via the video's DASH manifest. The request from the client was passed through the L3BOU edge server and forwarded to the video server which then returned the downscaled ULR tiles of our choosing. Upon return, the *number of bytes* counter was added. Bandwidth data measurements are shown in Figures 5, 6, 7, 8, and 9. The bandwidth results include the bytes transferred for metadata such as the DASH manifest.

The results in Figures 5, 6, 7, 8, and 9 show the effect of different ULR tile resolutions at different ratios to the full quality tiles during streaming. The left-most bars of the graphs represent the scenario in which no ULR tiles were streamed (the NO-SR use case), while the right-most sides of the graphs represent the amount of data transferred when only ULR tiles are sent (the ALL-SR case). As stated in previous sections, the chosen tile sizes for our testing scenarios are 24x24 and 48x48 pixels. In addition, their file sizes are significantly reduced when compared to those of 96x96 tiles. This decreased file size results in a nearly two-thirds or 20 megabyte drop in transmitted data per video when using 24x24 tiles at a 100 percent tile prediction success rate, as shown in Figure 7. For the 48x48 tile selection case, the results show that the video has decreased in size by half of its original size, or 15 megabytes at a 100 percent tile prediction success rate.

In other non-ideal scenarios, where not all of the tiles are predicted correctly by the prefetching service, there is still the possibility of significant savings with regards to the streamed data. In the 24x24 tile size case, our system decreases the data transferred by 10 megabytes, or one third of the total video size at a 50 percent successful prediction rate. This decreased size is similar to the savings attained when using the 48x48 tile at a 60 percent success rate. This data can also be leveraged to estimate the needed network bandwidth required for the connection between the video server and edge server.

Using the experimental results found from our testing, we make two fundamental assumptions about our network. We assume that the edge to client connection has sufficient bandwidth to support the stream and maintains a sub-50 ms network latency, which was found to be the highest supportable latency for Virtual Reality 360°-like streams [26]. The total data, transferred when no tiles are of ULR format, results in 32.66 MB of network traffic. Our sample video has a playtime of 165 seconds and we will assume a worst case cloud to edge latency of 102 ms along with a best case latency of 13 ms [15].

Discussion: As each segment is one second long, we determine that 197.9 KB of data must be transferred per segment. We are already limited by the cloud to edge server's network latency range, so assuming a best case scenario from cloud to edge, our network latency would be 13ms. Next, we assume the edge to client connection has a bandwidth of 15 MB/s and a network latency of 10 ms [2] to the edge server. With these constraints, this leaves only 13.8 ms for data transfer between the cloud and edge, and as a result the required cloud to edge bandwidth would be 14.34 MB/s. In this case, as little as seventy viewers can saturate a gigabit connection. When these same metrics are applied to our L3BOU system, the following considerations can be taken into account. First, since we are prefetching and then upscaling, we have a buffer period of one segment length to perform operations on the data and fetch it. From the sections above, we found that our lowest acceptable upscaling time was 200ms, and through our experiments, the parsing, stitching, encoding and decoding per task took an additional 425ms. When the previous network latency is taken into account, this leaves over a third of a second, 338.8 ms to fetch the segment tiles from the cloud server. This means that the required bandwidth for L3BOU in this situation would be as low as 584.3 KB/s. This extra latency buffer could also be used to connect from the L3BOU edge to further cloud servers at the price of higher required bandwidth.

V. Conclusion

In this paper, we introduce L3BOU, a novel three-tier software architecture that significantly alleviates backhaul network congestion for cloud to edge 360° streaming scenarios. To accomplish this goal, L3BOU leverages Navigation Graph-based prefetching of downscaled video tiles packaged for DASH to decrease transmitted data size. Utilizing edge computing resources, L3BOU performs *optimized super resolution* algorithms on segmented tiles in a multi-processed fashion to be sent to the client. Our results show that with L3BOU, the bandwidth of the backhaul network decreases by up to a factor of 24, and we are able to connect to cloud servers that have a much larger inherent network latency. Furthermore, our study shows that the utilization of edge-based optimized super resolution expands the effective range of cloud-edge connections.

VI. Acknowledgement

This work was funded by National Science Foundation (NSF) grants CNS 19-00875 and CNS 19-01137. All opinions and statements in the above publication are of the authors and do not represent NSF positions.

References

[1] FFMPEG. https://www.ffmpeg.org/, 2019.

[2] Batyr Charyyev, Engin Arslan, and Mehmet Gunes. Latency comparison of cloud datacenters and edge servers. 12 2020.

[3] François Chollet et al. Keras. https://keras.io, 2015.

[4] Mallesham Dasari, Arani Bhattacharya, Santiago Vargas, Pranjal Sahu, Aruna Balasubramanian, and Samir R Das. Streaming 360-degree videos using super-resolution. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1977–1986. IEEE, 2020.

[5] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org.

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

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

[8] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Jean Le Feuvre. Gpac filters. In *Proceedings of the 11th ACM Multimedia Systems Conference*, page 249–254, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[12] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017.

[13] Zhenxiao Luo, Zelong Wang, Jinyu Chen, Miao Hu, Yipeng Zhou, Tom Z. J. Fu, and Di Wu. Crowdsr: Enabling high-quality video ingest in crowdsourced livecast via super-resolution. In *Proceedings of the 31st ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, page 90–97, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[15] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. 10 2018.

[16] Pantelis Maniotis, Eirina Bourtsoulatze, and Nikolaos Thomos. Tile-based joint caching and delivery of 360 videos in heterogeneous networks. *IEEE Transactions on Multimedia*, 22(9):2382–2395, 2019.

[17] Jounsup Park and Klara Nahrstedt. Navigation graph for tiled media streaming. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 447–455, 2019.

[18] Jounsup Park, Mingyuan Wu, Kuan-Ying Lee, Bo Chen, Klara Nahrstedt, Michael Zink, and Ramesh Sitaraman. Seaware: Semantic aware view prediction system for 360-degree video streaming. In *2020 IEEE International Symposium on Multimedia (ISM)*, pages 57–64, 2020.

[19] Sung Cheol Park, Min Kyu Park, and Moon Gi Kang. Super-resolution image reconstruction: a technical overview. *IEEE signal processing magazine*, 20(3):21–36, 2003.

[20] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[21] Nikolaos Thomos, Nikolaos V Boulgouris, and Michael G Strintzis. Optimized transmission of jpeg2000 streams over wireless channels. *IEEE Transactions on image processing*, 15(1):54–67, 2005.

[22] Marko Viitanen, Ari Koivula, Ari Lemmetti, Arttu Ylä-Outinen, Jarno Vanne, and Timo D. Hämäläinen. Kvazaar: Open-source hevc/h.265 encoder. In *Proceedings of the 24th ACM International Conference on Multimedia*, 2016.

[23] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.

[24] Chenglei Wu, Zhihao Tan, Zhi Wang, and Shiqiang Yang. A dataset for exploring user behaviors in vr spherical video streaming. In *Proceedings of the 8th International Conference on Multimedia Systems*, MMSys '17, Taipei, Taiwan, 2017. ACM.

[25] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 645–661, 2018.

[26] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. Towards efficient edge cloud augmentation for virtual reality mmogs. 10 2017.

[27] Michael Zink, Ramesh Sitaraman, and Klara Nahrstedt. Scalable 360° video stream delivery: Challenges, solutions, and opportunities. volume 107, pages 639–650, 2019.