Semantic-Aware View Prediction for 360-Degree Videos at the 5G Edge

Shivi Vats
Dept. of Information Technology
University of Klagenfurt
Klagenfurt, Austria
shivi.vats@aau.at

Jounsup Park
Dept. of Electrical and Computer Engineering
California Baptist University
Riverside, CA, USA
joupark@calbaptist.edu

Klara Nahrstedt
Dept. of Computer Science
University of Illinois
Urbana-Champaign, IL, USA
klara@illinois.edu

Michael Zink
Dept. of Electrical & Computer Engineering
University of Massachusetts
Amherst, MA, USA
zink@ecs.umass.edu

Ramesh Sitaraman
College of Information & Computer Sciences
University of Massachusetts
Amherst, MA, USA
ramesh@cs.umass.edu

Hermann Hellwagner
Dept. of Information Technology
University of Klagenfurt
Klagenfurt, Austria
hermann.hellwagner@aau.at

2022 IEEE International Symposium on Multimedia (ISM) | 978-1-6654-7172-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISM55400.2022.00025

***Abstract*—In a 5G testbed, we use 360° video streaming to test, measure, and demonstrate the 5G infrastructure, including the capabilities and challenges of edge computing support. Specifically, we use the *SEAWARE* (*Semantic-Aware View Prediction*) software system, originally described in [1], at the edge of the 5G network to support a 360° video player (handling tiled videos) by view prediction. Originally, SEAWARE performs semantic analysis of a 360° video on the media server, by extracting, e.g., important objects and events. This video semantic information is encoded in specific data structures and shared with the client in a DASH streaming framework. Making use of these data structures, the client/player can perform view prediction without in-depth, computationally expensive semantic video analysis. In this paper, the SEAWARE system was ported and adapted to run (partially) on the edge where it can be used to predict views and prefetch predicted segments/tiles in high quality in order to have them available close to the client when requested. The paper gives an overview of the 5G testbed, the overall architecture, and the implementation of SEAWARE at the edge server. Since an important goal of this work is to achieve low motion-to-glass latencies, we developed and describe "*tile postloading*", a technique that allows *non*-predicted tiles to be fetched in high quality into a segment already available in the player buffer. The performance of 360° tiled video playback on the 5G infrastructure is evaluated and presented. Current limitations of the 5G network in use and some challenges of DASH-based streaming and of edge-assisted viewport prediction under "real-world" constraints are pointed out; further, the performance benefits of tile postloading are disclosed.**

***Index Terms*—Tile-based 360° video streaming, viewport prediction, tile postloading, 5G networks, edge computing**

I. Introduction

360° videos are a rich and interactive way to consume media. For high-quality content and, ideally, an immersive user experience, high-resolution imagery, substantial downlink data throughput, and very low motion-to-glass latency are required. Previous research efforts have proposed and developed DASH- and tile-based 360° video streaming systems and software as viable solutions. There also seems to be a consensus that (accurate) future view prediction is necessary to address the low latency problem and improve the quality of experience (QoE). Furthermore, edge computing is regarded as beneficial for meeting low-latency requirements.

SEAWARE [1] is a recent approach and software system for future view prediction in 360° videos to improve users' QoE. SEAWARE works on top of DASH and uses a video's semantic information and users' viewing patterns to perform view prediction at the client. Two new data structures, the *Semantic Flow Descriptor* (*SFD*) and the *View-Object State Machine* (*VOSM*) are introduced, containing semantic and user behaviour information, respectively. The tiled 360° video is analysed (offline) on the media server using deep learning mechanisms, and the generated data structures are stored alongside the video. Whenever the client requests video data, this additional information is transmitted to the client utilising an advanced *Media Presentation Description* (*MPD*). Thus the client can perform semantic-aware view prediction for a number of future segments (the prediction horizon of 'k' seconds) without analysing the video data itself. SEAWARE performs better than regular history-based view prediction algorithms and showcases the advantages of utilising video semantic data for view prediction.

In this work, we combine the SEAWARE approach to view prediction and the edge computing paradigm and develop an edge-assisted on-demand 360° video streaming system. To achieve this, we ported (part of) the SEAWARE software to run on the edge and perform view prediction and tile prefetching there. In the current configuration, the edge server acts as the "client" in a typical SEAWARE-based streaming system. In technical terms, this means that the edge server obtains the SFD and VOSM from the media server and performs future view prediction on behalf of the client. The results of the view prediction are used to prefetch and cache tiles on the edge server, allowing them to be transmitted with minimal latency if/when the client requests them in the future. To further reduce the motion-to-glass latency, we modified the 360° video player

978-1-6654-7172-5/22/$31.00 ©2022 IEEE
DOI 10.1109/ISM55400.2022.00025

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.

of [2] to implement "*tile postloading*", allowing the player to react to viewport changes as soon as they occur. To the best of our knowledge, such a technique has not been used for 360° video streaming before, but turned out to be very beneficial on unexpected (mispredicted) viewport changes.

The contributions of this paper are as follows:

- The SEAWARE system is (partially) ported to an edge instance and integrated into a fully functional 5G-, DASH-, and tile-based on-demand 360° video streaming system.
- Tile postloading is implemented in the 360° video player, allowing for quicker viewport/tile updates at the client and leading to lower motion-to-glass latencies.
- Importantly, the modifications made to the client and the system implemented at the edge work independently from each other; either component can be placed as-is into the streaming system.
- The performance of the SEAWARE system at the 5G edge is analysed and presented. Some limitations of the current 5G network and challenges of edge-based view prediction under "real-world" constraints are pointed out.

The remainder of this paper is organised as follows. Sect. II discusses related work. The SEAWARE system at the 5G edge and tile postloading are described in Sect's. III and IV, respectively. Sect. V discusses the dataset and the evaluation setup and presents the performance results. Sect. VI concludes the paper.

II. Related Work

Streaming 360° videos consumes a significant amount of bandwidth [3]. However, technologies such as HEVC [4] and DASH [5] can help combat the bandwidth requirement. Tiled video encoding as part of HEVC splits the 360° video into independently processed rectangular blocks known as *tiles*. Encoding tiles outside the user's viewport in low quality leads to bandwidth savings [6]. However, when a user changes their viewport, the tiles in the new viewport need to be updated to high quality as quickly as possible.

Hence, viewport prediction algorithms such as SEAWARE [1] were developed to predict the user's future viewport in advance and prefetch those tiles to the client. These algorithms must be accurate to provide a good QoE to the user [7]. However, performing viewport prediction on the client requires significant computational power, which is often limited on mobile clients.

Edge computing is based on the concept of placing computational and storage resources close to the client [8]. This enables low-latency responses and real-time data transfer to the client. This can be especially helpful in the context of latency-sensitive media such as 360° videos. Furthermore, tasks can be offloaded from the client to an edge instance, allowing client devices with limited computational power to obtain results of computationally intensive tasks.

Hence, various edge-assisted 360° video streaming techniques have been researched. Caching 360° video content at the edge allows it to be accessible with minimal latency compared to fetching it from the CDN. Various caching

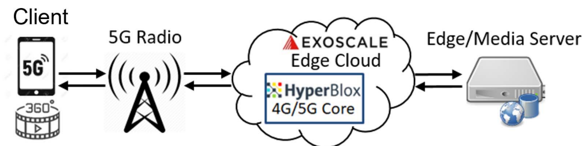


Fig. 1. 5G Testbed

techniques have been developed with a focus on optimising the storage overhead of the cache on the edge while still serving enough video data from the cache to make it beneficial [9].

A joint caching and computation system was also developed, utilising both the caching capability and computational power of the edge [10]. A computationally-intensive projection task could be performed at the edge instance to compress the viewport data to be sent to the client device. While leading to bandwidth savings, the client device would still have to decompress the viewport before displaying it. However, the client device could cache either the compressed or decompressed viewport. Caching the decompressed viewport would reduce the need to decompress it before display, but caching the compressed viewport would lead to a lower storage overhead. Hence, efficiency is important for such an implementation.

III. SEAWARE at the 5G Edge

A. 5G Infrastructure

Our 5G testbed is depicted in Fig. 1. An interesting feature of this 5G infrastructure – available within the project *5G Playground Carinthia*[1] and provided by the mobile network operator *A1*[2] – is that the 5G core functionality is fully implemented in software, provided by a product by *HyperBlox Inc.*[3] running in an *Exoscale*[4] edge cloud. The primary purpose of the infrastructure is to explore the capabilities and challenges of (software-based) 5G networks. For reasons of ease of system administration, our edge server is located on a separate machine, co-located with (but logically completely separate from) the media server. Basic measurements (*ping* and *iperf*) show that, at the time of writing, the one-way latency between the edge server and the client is < 6 ms on average, the downlink throughput is > 500 Mbps, and the uplink throughput is > 100 Mbps. We use a 5G-enabled Android smartphone slid into a VR headset as the client/playout device. All experiments were performed indoor in the 5G Playground premises, with the client device a few tens of metres away from the 4G/5G antenna, yet not in the same room.

B. Functionalities of SEAWARE at the Edge

The system at the edge contains three major functionalities:

[1] https://5gplayground.at/
[2] https://www.a1.net/, https://www.a1.group/
[3] http://hyperblox.io/
[4] https://www.exoscale.com/

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.

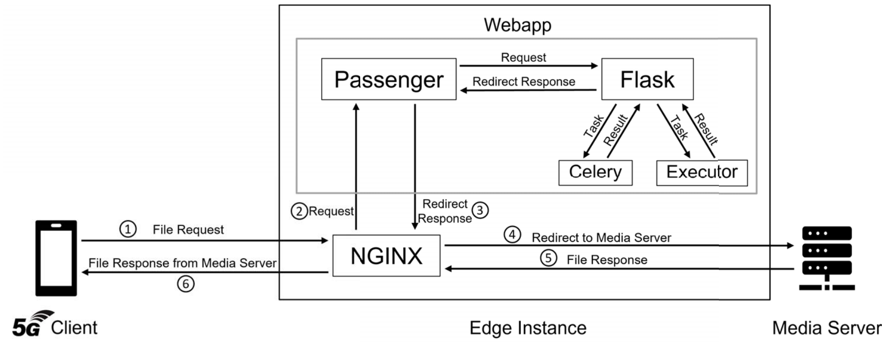


Fig. 2. Overall Workflow of the SEAWARE System

1) Serving On-demand 360° Videos to the Client: The videos that need to be served are stored on the media server. The videos are publicly available and can be streamed regularly by a suitable client. However, our client connects to the edge instance to utilise the SEAWARE system. To make the media server contents accessible via the edge, we deploy an *NGINX*[5] web server on the edge instance that acts as a reverse proxy. When the edge instance (reverse proxy) receives a request from the client, the reverse proxy can process it and forward it to the media server. The media server will then send the response to the reverse proxy, which will, in turn, send it back to the client. Fig. 2 shows the overall workflow of the SEAWARE system. On top of reverse proxy functionality, NGINX has a powerful caching system which we utilise to serve files from the edge instance with minimal latency.

2) Executing SEAWARE with Low Latency: The advanced MPD generation part of SEAWARE, including creating and storing the SFD and VOSM data structures, still runs on the media server side in our system. In a typical SEAWARE system, the advanced MPD would then be sent from the media server to the client, which would execute the semantic-aware view prediction component. The key difference in our implementation is that the view prediction algorithm of SEAWARE is offloaded from the client to the edge instance.

To execute SEAWARE on the edge instance, we utilise a Python web application, also known as *webapp*; see below. A Python port of the original SEAWARE view prediction algorithm was created and integrated into the webapp.

SEAWARE at the edge needs the user's current viewport as an input to the view prediction algorithm. This is obtained by analysing the requests sent from the client. The tiles stored on the media server are named according to a segment template specified in the MPD file of a video. In our case, the template is `seg-strQ-demuxedT-S.m4s`, where Q, T and S are the quality level (as an integer), the tile number, and the segment number, respectively. The client utilises this template to form requests. Since the system at the edge knows which files the client requests, it can determine the user viewport by analysing the requests and identifying the high-quality tiles requested for every segment.

3) Preloading of Tiles Belonging to Predicted Viewports: The result of the semantic-aware view prediction algorithm is a list of visible tiles in the future viewports for the user. SEAWARE at the edge converts the results into filenames using the naming scheme from the MPD. With the help of NGINX, these files are then fetched from the media server and stored in the NGINX cache. When NGINX receives a file request from the client, the cache is checked first for the file. A file found in the cache is served from it, avoiding forwarding the request to the media server and waiting for a response.

C. Implementation

The components of SEAWARE at the edge are as follows:

1) NGINX: NGINX is configured mainly using directives from the `ngx_http_proxy`[6] module. The `proxy_pass` directive is utilised to forward the client's requests to the webapp. The `proxy_cache` family of directives is used to cache files in advance and to serve requests from the cache when applicable. To ensure that the cache has no significant storage overhead, files are stored in the cache for only one minute before being evicted.

2) Webapp: Flask[7] was chosen as the webapp framework of choice. It is used to process an incoming request from the client and send a response back. It is important to process the request as quickly as possible to minimise the latency overhead introduced by the webapp. Hence, only minimal processing is done on the main thread.

When Flask receives a request forwarded to it from NGINX, the nature of the file requested is checked. An MPD file being requested signifies the beginning of video playback. Hence, some operations are performed to prepare the webapp for new video playback. Most importantly, a function is executed asynchronously using Flask-Executor[8] to obtain and load the SFD and VOSM data structures corresponding to the video. Flask-Executor is a Flask extension that implements Python's built-in library for executing functions asynchronously.

[5]https://nginx.org/

[6]http://nginx.org/en/docs/http/ngx_http_proxy_module.html
[7]https://flask.palletsprojects.com/en/2.0.x/
[8]https://flask-executor.readthedocs.io/en/latest/

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.

If the file request is for a tile, some crucial data is extracted from the filename, and a task is submitted to Flask-Executor for further processing, including obtaining relevant data from the filename as explained earlier. When all the data for the user's viewport for a particular segment is obtained, the semantic-aware view prediction algorithm is executed using Celery[9]. This is a lightweight framework that also allows for the execution of tasks in the background, similar to Flask-Executor. However, we utilise Celery in this case since it supports task chaining, i.e., a task can be configured to execute when a result from another specified task is obtained. More specifically, as soon as the future viewports are obtained from the view prediction algorithm, another task is executed to initiate the caching of the tiles corresponding to the future viewports. The tiles to be cached are retrieved from the media server in advance.

Finally, Passenger[10] was utilised as the application server of choice to deploy the webapp. The app server, amongst other things, is responsible for managing requests sent to the webapp. Furthermore, Passenger and NGINX have out-of-the-box compatibility allowing NGINX to seamlessly forward incoming requests to Passenger.

IV. Tile Postloading

This section focuses on the need for and the concept of tile postloading, a term given to a set of modifications made to the *Tiled Player* app that we use [2].

A. Original Player App Design and Need for Modifications

Before describing tile postloading, it is essential to present how the original app functions. The app is designed to stream on-demand 360° videos using DASH. Videos are requested on a per-segment basis and stored in a local buffer. The buffer always holds the current and next segments to be played. The capacity of the playback buffer is the driving parameter for new segment requests. When the app notices that the buffer contains less than two segments, it sends requests to the media server for the next segment, as detailed below.

Tiles outside the user's viewport at the time of the request are requested in low quality; tiles inside the user's viewport are requested in the highest quality allowed by the network conditions. The tiles are downloaded individually, multiplexed into a segment, and then added per frame to the buffer. For a 30 fps video and 1-second segment length, for instance, a new segment would lead to 30 new entries in the playback buffer, each containing tile data for one frame. Hence, the buffer acts as a queue. As a consequence, when the user's viewport changes, the tiles in the new viewport are only obtained in high quality whenever the next segment request happens. Although in line with DASH specifications, this design has a significant drawback for our use case.

Let S_n be the nth segment of a 360° video, and F_n be the set of frames belonging to S_n. Let the user perform one head movement (the 'motion') when S_n is playing. Since the app maintains a local buffer of two segments, the frames from S_{n+1} already exist in the playback queue, having been requested before the motion. Thus, the high-quality tiles in F_{n+1} correspond to the user's previous viewport and might contain low-quality tiles in the user's new viewport. When S_n finishes playing, the app recognises that there is only one segment in the buffer and requests S_{n+2}. The app uses the user's viewport at the time of the request, and thus, the tiles in the frames belonging to F_{n+2} are assigned high-quality appropriately. When S_{n+1} is played, some tiles in the user's viewport are still low quality. However, when S_{n+2} plays, all the tiles belonging to the user's viewport are finally displayed in high-quality.

The motion-to-glass latency here would equal at least one segment length. Regardless of when the head movement happened in S_n, the user would always have to wait for S_{n+1} to finish playing for the updated viewport to be displayed to them. Since the main aim of this work was to minimise motion-to-glass latency, we found this design to be counterproductive. Ideally, this latency would be less than 20 ms to provide maximum comfort to the user [11]. Since we encode our videos with a 1-second segment length, a motion-to-glass latency of > 1 second was very undesirable.

B. Tile Postloading

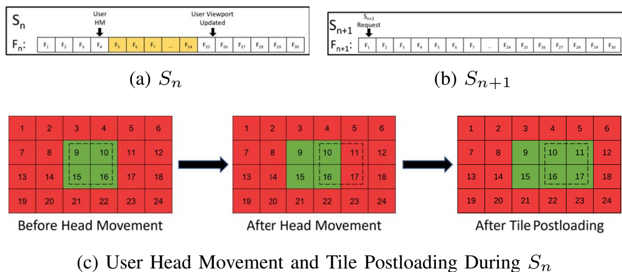
With tile postloading enabled, the app can request updated viewports for all segments already existing in the buffer, including the one currently being played back. The user's viewport changes can be reflected in the current segment, and the need to wait for the current or the following segment to finish playing is eliminated. In other words, the observed motion-to-glass latency is reduced significantly. Tile postloading is explained in detail with the help of the example from the previous subsection, illustrated in Fig. 3. Fig's. 3a and 3b depict the various events that occur during the playback of S_n and S_{n+1}, respectively. In Fig. 3c, the user's viewport is depicted using dashed lines. The tiles shown in red are of low quality, whereas the green ones are of high quality.

Let $T_{n,1}$ to $T_{n,24}$ represent the 24 tiles belonging to segment S_n. When the user performs a head movement during the playback of S_n, the app detects this movement, and the app immediately sends a request to update all the segments in the buffer, i.e., S_n and S_{n+1}. These updates are performed by modifying the tiles belonging to all frames in F_n and F_{n+1} to reflect the new viewport.

Say the user's viewport initially contains the tiles $T_{n,9}$, $T_{n,10}$, $T_{n,15}$, and $T_{n,16}$. Furthermore, say the user rotates their head to their right, leading to the new viewport containing tiles $T_{n,10}$, $T_{n,11}$, $T_{n,16}$, and $T_{n,17}$. To perform tile postloading, first, the tiles in the new viewport that are not already high quality are identified. Then a partial segment is requested, containing only these tiles but in high quality. In this example, a partial segment containing only tiles $T_{n,11}$ and $T_{n,17}$ will be requested for S_n. Similarly, a partial segment containing tiles $T_{n+1,11}$ and $T_{n+1,17}$ will be requested for S_{n+1}, as it also exists in the buffer. Let $T^{'}_{n,11}$ and $T^{'}_{n,17}$ represent the newly

[9]https://docs.celeryproject.org/en/stable/getting-started/introduction.html
[10]https://www.phusionpassenger.com/

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.



(a) S_n

(b) S_{n+1}

(c) User Head Movement and Tile Postloading During S_n

Fig. 3. Motion-to-Glass Latency Example with Tile Postloading

obtained high-quality tiles for S_n. These tiles then replace the already existing $T_{n,11}$ and $T_{n,17}$ in S_n, and are multiplexed with the rest of the tiles to obtain $S_n^{'}$, the updated segment with all $F_n^{'}$ containing high-quality tiles in the new viewport. Similar updates are also conducted for the rest of the segments already existing in the buffer. In this case, S_{n+1} would be updated with $T_{n+1,11}^{'}$ and $T_{n+1,17}^{'}$ to obtain $S_{n+1}^{'}$.

V. Performance Evaluation

This section focuses on the performance of our system. The dataset, the setup utilised to evaluate the system, and performance evaluation results are presented.

A. Dataset

We utilised the same dataset as in the original SEAWARE paper. The dataset [12] contains view traces from 48 users watching nine videos. The videos need to be prepared before they can be streamed. The preparation involved encoding the video into tiles and preparing an advanced MPD for the videos. Tiling the videos was achieved with the help of FFmpeg[11], Kvazaar[12], tileMuxer [2], and MP4Box[13]. The videos were encoded with a 1-second segment length, a `6x4` tiling pattern and three quality representations (low, medium, and high).

After the configuration, the SFD and VOSM files of the videos were placed in their root folder. These data structures were obtained from the original SEAWARE implementation. A Python script was utilised to add new elements named "SFD" and "VOSM" to the manifest, containing only one attribute named "path". The path of the SFD and VOSM files relative to the manifest file is set as the value of the respective "path" attributes. When the webapp detects a video MPD is requested by the client (tiled player), it can obtain the SFD and VOSM by reading these attributes in the manifest file.

Finally, the user view traces from the dataset were also prepared so that the player could perform hands-free testing by reading the view data and changing viewports autonomously.

B. Evaluation Setup

The evaluation setup contains various components. Some of these components are static, while others are changed to obtain different evaluation configurations.

[11]https://ffmpeg.org/
[12]https://github.com/ultravideo/kvazaar
[13]https://github.com/gpac/gpac/wiki/MP4Box

The videos are streamed over the testbed of Fig. 1 to a Samsung Galaxy S20 5G phone[14]. Various configurations are studied: from the media server or the edge server, with tile postloading or without, and utilising Wi-Fi or 5G.

Further, three prediction horizons (`'k'`) are utilised when streaming with SEAWARE: 1, 2, and 5 seconds. Results with `k=1` and `k=5` are compared to the original SEAWARE system. `k=2` is utilised in this paper due to the nature of our client app (player). The player requests one segment in advance using the user's viewport from the currently viewed segment. When considering file requests, the initial viewport requested for any immediate next segment would be the same as the viewport for the current one. Hence, predicting the immediate next segment with our SEAWARE system will not significantly impact the system's performance, as the full segment would be requested before any view prediction is performed.

Finally, random viewport changes and user view traces are utilised for autonomous testing with the player.

C. Motion-to-Glass Latency

The system times at the time of a viewport change ('motion') and at the time of frames being displayed ('glass') are logged in the client app (tiled player). These logs contain information regarding tiles in the viewport and their quality. The motion-to-glass latency is calculated by taking the difference in the system times between the viewport change and all the visible tiles displayed in high-quality. Notably, any cases where the visible tiles after a viewport change already existed in high quality beforehand are not considered for motion-to-glass latency calculation.

Furthermore, a metric called the "success rate" is introduced. There could be a scenario that the viewport would never be updated with high-quality tiles due to poor network conditions before another viewport change happened. The success rate represents the percentage of viewport changes successfully met with a high-quality viewport.

1) Playback after Random Viewport Changes: Fig. 4 shows the comparison of Wi-Fi and 5G with both regular playback of the videos and playback with tile postloading enabled, under random viewport changes. It can be observed that both networks perform similarly for regular playback. This is expected, as the motion-to-glass latency is a minimum of one segment length (1 second in our case), and that is enough time for both networks to download future viewports.

For playback with tile postloading, 5G performs 39.25% worse than Wi-Fi on average. Since the viewports are changed randomly, many tiles can be added to the viewport as part of one change. The 5G network does not perform as well as the Wi-Fi network in downloading many tiles simultaneously.

The reason for the 5G network's performance is twofold. Firstly, the 5G network we use is operated in non-standalone (NSA) mode, i.e., it utilises the existing 4G LTE infrastructure. Hence, the client device often switches to 4G even when

[14]Model number SM-G918B/DS; http://www.samsung.com/us/mobile/galaxy-s20-5g/specs/

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.

(a) Regular Playback

(b) Playback with Tile Postloading Enabled

Fig. 4. Playback after Random Viewport Changes: Motion-to-Glass Latency

Fig. 5. Playback after Random Viewport Changes: Success Rate

it is supposed to utilise 5G. Secondly, we measured the performance of our 5G and Wi-Fi networks using iperf3[15], curl[16], and termux[17]. While both connections had adequate bandwidth, the downloading times for a 32.9 kB file were significantly different. On 5G, it took the client device 1.15 and 1.09 seconds to download the file from the media server and the edge cache, respectively. However, the times measured with Wi-Fi were only 0.22 and 0.05 seconds. Hence, the Wi-Fi performed significantly better than the 5G network in downloading files. Furthermore, for Wi-Fi, tile postloading led to an average 62.59% lower motion-to-glass latency; for 5G, the improvement was 48.06% on average. Hence, Wi-Fi has more potential to take advantage of tile postloading than the 5G network currently in use.

Fig. 5 shows the success rate of the four configurations under random viewport changes. The success rate is very high for all configurations, with an average of 94.29%. Hence, the network conditions were adequate regardless of the network being utilised and tile postloading being enabled.

2) Playback with User View Data: Fig. 6a shows the motion-to-glass latency results for our system when using the prepared user view traces. These measurements were done only with the 5G network since the focus here was to compare the various SEAWARE configurations instead of the networks.

It can be noted that the latencies are similar for all four configurations. This is not the expected result, as the SEAWARE system at the edge should reduce the latency due to some files being served from the cache at the edge instance.

[15]https://iperf.fr
[16]https://curl.se/
[17]https://termux.com/

(a) Motion-to-Glass Latency

(b) Success Rate

Fig. 6. Playback with User View Data (5G Network)

There are a few reasons for this result. Firstly, the 5G network is not as stable and well-performing as expected, as described in the previous subsection. Even though enough relevant head movements were obtained to compensate for the unstable nature of 5G, a simple test showed that the latencies were lower when using Wi-Fi. The motion-to-glass latency for video 3 with `k=5` was 216 ms when streaming with Wi-Fi and 492 ms for streaming with 5G. Furthermore, the media server in our configuration was still only a few hops away from our client device, i.e., close enough that streaming directly from the media server also yielded good results.

Finally, the performance of our SEAWARE system, as discussed in the following subsection, also impacts the motion-to-glass latency. To achieve low motion-to-glass latency, all the tiles included in a request must be served from the NGINX cache. If even one of the tiles needs to be fetched from the media server, then the latency will be high, as the client device needs to wait for that tile to fulfil the overall file request. Our system apparently cannot consistently preload full viewports needed by the client from the NGINX cache. However, when the whole viewport is found, the latency is significantly lower, as observed by the client recording download times of 20–40 ms when only one or two tiles were requested.

Fig. 6b shows the success rate when using the prepared user view traces. With a 96.28% average, the success rate is very high. Hence, in the case of streaming with user viewport data, the network is still adequate for the client to load high-quality viewports in response to the head movements.

D. SEAWARE Performance and Storage Overhead

The view prediction accuracy of SEAWARE plays a crucial part in determining the system's overall effectiveness since only tiles predicted by the SEAWARE algorithm are cached by NGINX at the edge. We measure the metrics related to the view prediction performance of our SEAWARE system by using modified versions of MATLAB scripts from the original SEAWARE implementation. Furthermore, since we utilise the same user view traces and algorithms as the original SEAWARE system, its performance is compared with the performance of our SEAWARE system.

Fig's. 7 and 8 show the precision and prediction error metrics, respectively, of our SEAWARE system. On average, our SEAWARE system has a precision of 0.733 and 0.623 for `k=1` and `k=5`, respectively. Likewise, our SEAWARE system

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.

(a) k = 1 (b) k = 5

Fig. 7. SEAWARE Precision Comparison

(a) k = 1 (b) k = 5

Fig. 8. SEAWARE Prediction Error Comparison

has an average prediction error of 0.175 and 0.259 for k values 1 and 5, respectively. Notably, a lower prediction error signifies better performance. Our system performs worse than the original system in all cases. More specifically, the precision of our system is 18.24% and 24.08% worse for k values of 1 and 5, respectively. Likewise, in terms of prediction error, our system is 12.21% and 15.95% worse for k values of 1 and 5, respectively.

Since the SEAWARE portion of both systems is identical to each other, this performance change can be attributed to the fact that our system operates in a "real-world" scenario, with certain constraints, whereas the original SEAWARE system is a simulation written in MATLAB. Our system utilises an Android client and various servers for streaming. The client device is not capable of performing more than one viewport change per second without introducing further processing lags. Hence, we modified the user viewport data such that our client would perform only one viewport change per second, whereas the simulated SEAWARE system reads the full view traces, about 90 per second. This allows the simulated system to develop a more accurate idea of the user's viewport. This viewport is used as input to the viewport prediction algorithm and to measure the performance by comparing it to the predicted viewports. Thus, the predicted viewports and the performance metrics calculated would be much more accurate for the simulated system.

Finally, the SFD and VOSM files generated for SEAWARE have an insignificant impact on the overall video size, respectively measuring at 0.0104% and 0.0009% of the total storage size of the videos on average.

E. NGINX Performance

NGINX plays a crucial part in our SEAWARE system, and the performance of NGINX is directly tied to the system's

(a) NGINX Cache Hit Rate (b) NGINX Response Time

Fig. 9. NGINX Performance

performance as a whole. We define two metrics to measure this performance: the NGINX cache hit rate and the NGINX request-response time.

1) NGINX Cache Hit Rate: On top of the precision and prediction error, the NGINX cache hit rate is another metric used to measure the performance and effectiveness of our system's SEAWARE view prediction algorithm. The NGINX cache hit rate is the percentage of file requests served from the cache compared to the total number of file requests. This hit rate represents the "real-world" performance of the system, as it directly represents the number of files served to the user from the NGINX cache with minimal latency.

Fig. 9a shows the NGINX cache hit rate measurements for our SEAWARE system with `k=2` and `k=5`. The hit rate for `k=1` was 1.63% on average for all the videos. The low hit rate is due to the buffer management in the client app. When segment S_n is playing, S_{n+1} already exists in the buffer and S_{n+2} is requested. But with `k=1`, viewport prediction leads to tiles for S_{n+1} being predicted. In this case, the segment requested is not the one for which tiles are preloaded. Hence, the hit rate for `k=1` is insignificant and is consequently not plotted in the graph.

The average cache hit rates for `k=2` and `k=5` were 70.59% and 80.35%, respectively. The hit rate for `k=5` was on average 9.76% higher than for `k=2`. This increase is simply due to there being more tiles predicted and thus preloaded when the system is predicting the user viewport for the next five segments compared to the next two segments. More tiles in the NGINX cache lead to a higher probability of a tile requested by the client to be found in the cache.

2) NGINX Request-Response Time: The NGINX request-response time, or simply response time, represents how long our system takes to process the request and send a response back to the client. This time includes delays induced by any processing done in our system, such as delays induced by the webapp and the time needed to fetch a file from the media server. Notably, the response time is only calculated for high-quality tile requests, as those are the only tiles our system predicts and preloads into the cache.

Fig. 9b shows the NGINX request-response time for our SEAWARE system with all three k values. Unlike the cache hit rate, the response time for `k=1` is included in this graph since it could be plotted without skewing the scale too much for the other two configurations. Nonetheless, it can be noted that the

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.

response time for k=1 is significantly higher. This is due to only a small amount of requests being served from the NGINX cache in this case. A file being served from the NGINX cache means that NGINX does not have to communicate with the rest of the system or the media server and subsequently wait for their responses. Hence, a cache hit reduces the NGINX response time significantly. Since most files are not found in the NGINX cache for k=1, the average response time is significantly higher.

The average response times for k=2 and k=5 were 37 ms and 25 ms, respectively. The response times are lower due to the higher cache hit rate for the two configurations. Similarly, the response time for k=5 is lower than k=2 since the former had a higher cache hit rate than the latter.

Notably, the request-response time in NGINX for files served from the cache is 0.000 seconds. Since NGINX logs to millisecond precision, the response time for these requests is less than 1 ms. This is significantly smaller than the response time for files not served from the cache, which ranges from approximately 50 ms to over 200 ms.

F. Webapp Performance and Storage Overhead

The webapp is designed to execute most tasks off the main thread, adding minimal overhead to the request-response time. More specifically, the webapp adds 0.09 ms on average, a negligible amount compared to the overall NGINX response time. Another important metric is the time taken to perform view prediction using SEAWARE. On average, across all videos, the webapp takes 18.21 ms, 31.17 ms, and 71.04 ms to perform semantic-aware view prediction for k=1, k=2, and k=5, respectively. Furthermore, the time taken to send the view prediction results to NGINX is 12.82 ms, 24.45 ms, and 57.11 ms for prediction horizon values of k=1, k=2, and k=5, respectively. These times are much shorter than the 1-second segment length, ensuring that view prediction for a segment can be performed in real time.

Finally, the storage overhead of the app is measured to be 270.9 MB, which is negligible compared to the capacities of servers that usually range in the hundreds of GBs.

VI. Conclusion and Future Work

In this work, we developed a "real-world" system for edge-assisted view prediction in on-demand streaming of tiled 360° videos. The system was designed to minimise motion-to-glass latency and involved the development of a webapp at the 5G Playground edge. SEAWARE was utilised as the view prediction algorithm of choice. Furthermore, tile postloading was implemented in the 360° video player, allowing the player to harness the system's potential further and reduce the motion-to-glass latency. Finally, 5G was utilised as the communication medium between the client device and the server it connects to.

We observe that streaming with our SEAWARE system does not significantly change motion-to-glass latency compared to streaming the video directly from the media server. Furthermore, our SEAWARE system performs worse than the original SEAWARE system in terms of precision and prediction error. However, utilising tile postloading leads to a significant reduction in motion-to-glass latency. Additionally, the 5G connection at the 5G Playground performs worse than the Wi-Fi connection. At this time, the Wi-Fi connection also has a higher potential to take advantage of tile postloading than the 5G connection. Finally, the overheads added by the webapp are found to be insignificant in terms of response time and storage.

Since the performance of the 5G network is a notable limitation of the system, future development in that field will directly benefit this work. Other opportunities in future work also exist in the form of a view prediction algorithm designed specifically to be utilised in an edge-assisted scenario and a player designed with tile postloading and edge-assisted view prediction in mind. An edge-assisted view prediction system could also be designed for live-streamed 360° videos.

Acknowledgments

This work was supported by the National Science Foundation under Grant No. CNS-1900875 and No. CNS-1901137 and by the 5G Playground Carinthia (https://5gplayground.at/).

References

[1] J. Park, M. Wu, K.-Y. Lee, B. Chen, K. Nahrstedt, M. Zink, and R. Sitaraman, "SEAWARE: Semantic Aware View Prediction System for 360-degree Video Streaming," in *Proc. IEEE Int.l Symposium on Multimedia (ISM)*, 2020, pp. 57–64.

[2] M. Graf, C. Timmerer, and C. Mueller, "Towards Bandwidth Efficient Adaptive Streaming of Omnidirectional Video over HTTP: Design, Implementation, and Evaluation," in *Proc. 8th ACM Multimedia Systems Conference (MMSys)*, 2017, p. 261–271.

[3] S. Afzal, J. Chen, and K. Ramakrishnan, "Characterization of 360° Videos," in *Proc. Workshop on Virtual Reality and Augmented Reality Network*, 2017, pp. 1–6.

[4] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, 2012.

[5] T. Stockhammer, "Dynamic Adaptive Streaming Over HTTP—Standards and Design Principles," in *Proc. 2nd ACM Multimedia Systems Conference (MMSys)*, 2011, pp. 133–144.

[6] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth, and M. Zhou, "An Overview of Tiles in HEVC," *IEEE Journal of Selected Topics in Signal Processing*, vol. 7, no. 6, pp. 969–977, 2013.

[7] H. Wang, V.-T. Nguyen, W. T. Ooi, and M. C. Chan, "Mixing Tile Resolutions in Tiled Video: A Perceptual Quality Assessment," in *Proc. 24th ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2014, pp. 25–30.

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[9] S. Sukhmani, M. Sadeghi, M. Erol-Kantarci, and A. El Saddik, "Edge Caching and Computing in 5G for Mobile AR/VR and Tactile Internet," *IEEE MultiMedia*, vol. 26, no. 1, pp. 21–30, 2018.

[10] Y. Sun, Z. Chen, M. Tao, and H. Liu, "Communications, Caching, and Computing for Mobile Virtual Reality: Modeling and Tradeoff," *IEEE Transactions on Communications*, vol. 67, no. 11, pp. 7573–7586, 2019.

[11] R. Yao, T. Heath, A. Davies, T. Forsyth, N. Mitchell, and P. Hoberman, "Oculus VR Best Practices Guide," *Oculus VR*, vol. 4, pp. 27–35, 2014.

[12] C. Wu, Z. Tan, Z. Wang, and S. Yang, "A Dataset for Exploring User Behaviors in VR Spherical Video Streaming," in *Proc. 8th ACM Multimedia Systems Conference (MMSys)*, 2017, pp. 193–198.

Authorized licensed use limited to: University of Massachusetts Amherst. Downloaded on July 30,2023 at 05:09:22 UTC from IEEE Xplore. Restrictions apply.