2023-01-05

Anti-Freeze: High-Quality Adaptive Live Streaming with Real-time Transcoder

Mehmuda, Asif Ali

UNIVERSITY OF CALGARY

Anti-Freeze: High-Quality Adaptive Live Streaming with Real-time Transcoder

by

Asif Ali Mehmuda

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA
JANUARY, 2023

Abstract

Video streaming constitutes more than 80% of Internet traffic [1], and the demand continues to rise as interactive video applications like virtual conferencing/collaboration as well as augmented/mixed reality emerge. Such interactive video applications pose a challenge for real-time video transcoding and streaming. Transcoding is a computationally intense process and if not performed efficiently it can result in unwanted delays, which further limit the Quality-of-Service (QoS) delivered by the streaming protocol. In this thesis, we aim to address the real-time challenge and propose Antifreeze, a complete end-to-end solution for real-time transcoding and streaming. We propose a machine-learning solution to estimate the transcoding time and resource requirement, which is used to direct our real-time transcoder to transcode video segments at the rate matching their frame rate. We complete the Antifreeze design with a novel quality adaptation algorithm that not only considers visual quality and bandwidth dynamics, but also transcoding time and necessary computing resource allocation. Our results show that Antifreeze significantly reduces the playback stalls and substantially improves the visual quality in interactive video streaming sessions under various bandwidth profiles.

Acknowledgements

I would like to express my gratitude to my supervisors Dr. Mea Wang and Dr. Diwakar Krishnamurthy for their guidance throughout my graduate studies journey.

I would also like to thank my examiners Dr. Steve Drew and Dr. Richard Zhao for taking the time to review my thesis.

I would like to thank the University of Calgary and Mitacs for all the opportunities they provided me with along with all the necessary financial support.

Most importantly, I would like to thank my grandfather Shamoon Ali, my grandmother Amna Habib, my mother Khatoon Banu, my father Altaf Hussain, my brother Afroz Ali for giving me the necessary life skill so that I am able to able to fight through the challenges of living in a new country.

Not to forget my friends, I thank Aayush Sood, Alhamd Khan, Abeer Khan and Yang from bringing levity to this long journey.

Dedicated to my parents and grandparents who raised me with all their love

Table of Contents

List of Figures

List of Tables

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
ABR	Adaptive Bitrate
AVC	Advanced Video Coding
CTU	Coding Tree Unit
CU	Coding Unit
DASH	Dynamic Adaptive Streaming Over HTTP
fps	Frames Per Second
GOP	Group of Pictures
HAS	HTTP Adaptive streaming
HEVC	High efficiency Video Coding
HLS	HTTP Live Streaming
HTTP	Hypertext Transfer Protocol
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MPD	Media Presentation Description
PSNR	Peak Signal-to-noise Ratio
PU	Prediction Unit
QoE	Quality of Experience

RMSE	Root Mean Square Error
SSIM	Structural similarity Index
VM	Virtual Machine
VMAF	Video Multimethod Assessment Fusion
VOD	Video on Demand

Chapter 1

Introduction

Video streaming is a major source of entertainment, information, and e-learning worldwide, and according to Cisco [1], video data will account for 82% percent of internet traffic by 2022. Although the network bandwidth is increasing [14, 15], the increase in streamed video volume is much larger. Improved network infrastructure coupled with advancements in video codecs and end-user devices have led to an exponential rise in streamed video traffic and promoted streams in the ultra-HD (4K) range and beyond.

A consumer watching a video stream will have a poor viewing experience if the quality of the video is bad, the video buffers (or stalls) for too long, or if there are long delays between when the video was requested versus when the video was visible to them as a viewer. Consequently, the objective of any video streaming service is to provide a *high Quality of Experience* (QoE) to the viewers by maximizing the visual quality and minimizing delays and stalls/buffering [16, 17]. These user experience metrics, namely visual quality, delay, and stalls, are competitive in nature, *i.e.,* improving one inherently results in the degradation of the other(s). For instance, the visual quality of the stream can be enhanced by streaming at the highest available quality level (bitrate) [17], but this puts the client at the risk of running out of content in case the bandwidth drops, resulting in playback stalls. To mitigate the effects of bandwidth drops, the client stores a few seconds of playback in the buffer (*i.e.,* its

memory) before beginning the playback. Having a large buffer results in smoother playback as it minimizes the chances of stalls, but on the other hand, it leads to increased delays since playback cannot begin till the client has fetched and stored enough segments in the buffer. Meeting these QoE expectations is an even bigger challenge for live streaming, where the content needs to be delivered as soon as it is generated. For a live stream, *glass-to-glass delay* or *glass-to-glass latency* is the time between when a video was recorded and when the client plays it [18]. A delay of a few seconds is acceptable for Video on Demand (VOD) applications such as Netflix and YouTube. However, live, and interactive applications such as online gaming and video conferencing are delay sensitive and need near real-time delivery.

1.1 Live Streaming Architecture

Fig. 1.1 illustrates the stages of a live streaming pipeline [19] and the components of the pipeline are described below:

- **Video Generation:** A video source (*e.g.,* camera) continuously captures a video as a sequence of digital frames. Live videos are transmitted as a sequence of segments, and the segments are generated periodically with a frequency of one segment every d seconds where d is the segment duration. A video encoder then compresses the video to reduce its file size while retaining as much visual quality as possible.

- **Media Server:** The video is then passed on to a media server which transcodes the video into multiple quality levels (transcoding stage), and packages it into a format suitable for delivery to the client (packaging stage).

- **Content Delivery Network (CDN):** The media server transmits the video to a CDN. A CDN is a group of servers that reduces the delay in transmitting the video to a consumer by ensuring that the video requested by a consumer is delivered by a server located geographically closer to the consumer [20] (edge server).

- **Video Player:** Finally, the video player fetches the video segments from the media server (or CDN) into its buffer (*i.e.,* its memory) in a sequential manner. After the player has stored sufficient segments in its buffer, it starts outputting the segments on its display and the video begins playing.

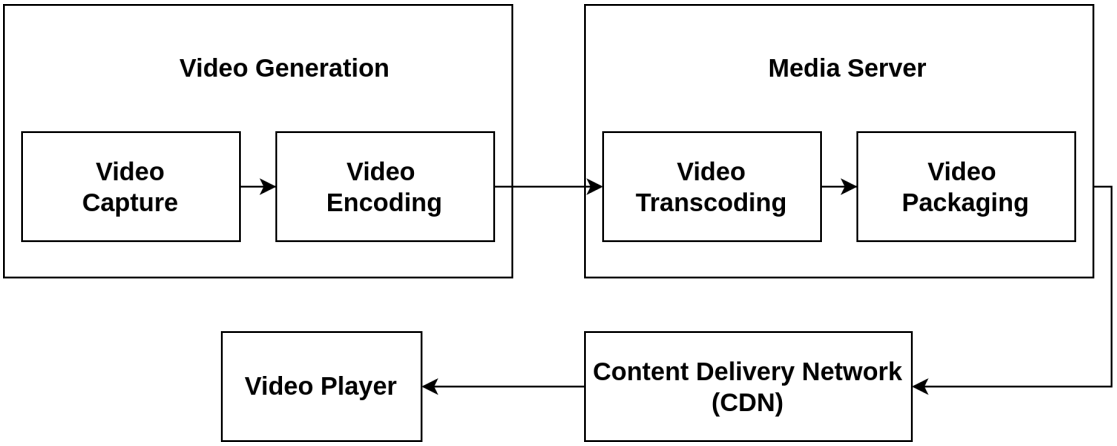


Figure 1.1: Stages of a live streaming pipeline

.

Each component of a live-streaming pipeline introduces delays. Specifically, the Media Server introduces a transcoding delay, the CDN and the network introduces a transmission delay, and the Video Player takes time to store enough video segments in its buffer (to avoid playback stalls) before beginning video playback introducing buffer delay. All these delays contribute to the overall *glass-to-glass* delay. A higher delay implies a reduced ability to deliver video content to an end user in real-time. Real-time delivery in the context of live streaming refers to minimizing the delay between when an event was captured versus when the viewer views it.

Transcoding a live stream prior to delivering it to the end users is crucial as it transforms the video into a format appropriate for the client to play given the device's capabilities and playback conditions. However, video transcoding is a complex process as it involves decoding a video and encoding it again with new properties [21, 22]. It is often the transcoding process, which decides if a live video can be streamed in real-time.

1.2 Live Streaming Challenges

Live streaming a video faces the following transcoding challenges:

- **Video transcoding has to be real-time:**. A key difference between video-on-demand (VOD) streaming and live video streaming is that in the former it is possible to pre-transcode and store the video segments at several quality levels (representations) whereas in the latter transcoding must be done on-the-fly in real-time. For sustaining a live stream, the video transcoding rate must be at least as high as the live video rate (video generation rate), or else the client will experience playback delays and interruptions.

- **Video transcoding is computationally demanding and resource-intensive:** The transcoding process is computationally intense, especially for high-resolution videos, and if sufficient resources (*i.e.,* CPU cores) are not provisioned, the transcoding rate will lag the live video rate. However, on the other hand, over-provisioning of transcoding resources (*i.e.,* CPU cores) can lead to an increase in operational cost for cloud-based providers. Hence making the transcoding rate meet the video rate while using just the right amount of resources is essential.

- **Video Transcoding has to be client adaptive:** The quality level to which the stream gets transcoded must match the viewer's needs and playback conditions to provide a high QoE to the viewers. Therefore the transcoder must be adaptive to the client's needs and must deliver the best quality segment to the client in real-time.

- **Video Transcoding has to be content adaptive:** There are a variety of video genres and so are the transcoding and delivery challenges they present. Additionally, not only do videos vary, but segments belonging to the same video may also differ in complexity [23]. Hence, the transcoder must also factor in the segment complexity when transcoding a stream in real-time.

In summary, we can state that delivering live content to the end users is a challenging process. Not only should the transcoding rate be fast enough so that the real-time nature of the live stream is not compromised, but the resources allocated to the transcoder must be optimally provisioned (neither under-allocation nor over-allocation) to achieve resource efficiency. Additionally, video transcoding must be adaptive to the client's needs and deliver optimal quality segments to the client to maximize the QoE.

To provide a high QoE to the end user, stream providers transcode a video stream into multiple quality levels [24] for an end-user to pick up based on the playback conditions (*i.e.,* bandwidth and buffer). However, this approach of transcoding a video to a set of static pre-defined quality levels forces the client to select a quality from among the offered quality levels and there may not always be a close quality match to what the client expects. For instance, if the video is transcoded at 3 quality levels - 500 kbps(Low quality), 2000 kbps (mid quality), and 5000 kbps (high quality) and the client has a bandwidth of 1500 kbps, the client must stream at the 500 kbps low quality level as that is the closest quality level which is lower than the bandwidth whereas it would have been optimal if the client could have streamed at a higher quality level close to its bandwidth.

There have been prior works on enhancing the QoE of a viewer. In particular, on-the-fly transcoding solutions can enhance the viewers QoE by dynamically adapting to the client's needs during run-time. However, for the proposed on-the-fly transcoding solutions [25, 26] the focus is on minimizing the costs associated with transcoding and not enhancing the viewers QoE. Similarly, many quality-aware ABR algorithms have been proposed in literature to enhance the viewers QoE [16, 27, 17, 28, 29], but the majority of them have been designed for Video on Demand (VOD) applications and not live streams which needs real-time transcoding. Considering transcoding overheads while dynamically transcoding a stream in real-time is a major research focus of this thesis.

Although transcoding a video stream to a fixed set of representations (bitrate resolution pairs) makes the streaming service more scalable as the transcoding server is agnostic to the

client playback conditions and it is the client which decides which of the offered representations best fits its needs. For such a fixed multi representation setup new clients can simply fetch the appropriate representation from the server without requiring any changes on the server side. However, this approach is not tailored to the needs of individual clients. In this thesis, we propose a solution to enhance QoE on a per-client basis by transcoding a video on the fly to the most optimal variant that the client can stream. The adaptive solution presented in this thesis has been designed to only entertain one client at a time (one-to-one communication). In the future, this architecture could be extended to handle requests from multiple clients. A possible approach to scale the system as multiple clients join the stream is to batch similar client requests (based on similarity) or process the requests in parallel along with horizontally scaling the number of servers. Our proposed solution has been designed on the application layer and does not require changes on the low-level protocols and thus can be used for both unicast and multicast applications (something to be implemented in the future). Within a CDN network, the video content generated by the live source can be quickly replicated to the edge servers and the edge servers can then dynamically transcode the videos based on the client needs. However, the scope of caching is limited since the transcoded content is tailored to the needs of individual clients rather than to serve a wider target audience. Nevertheless, the focus of this thesis is to evaluate the effectiveness of an on-the-fly adaptive streaming platform and proposing strategies to make the system scalable is out of scope of this thesis.

1.3 Research Objectives

The primary objective of our research is to maximize the QoE of a live HTTP stream by means of adaptive on-demand real-time transcoding. On-demand transcoding throughout the remainder of this thesis shall refer to transcoding performed by the server on an on-demand basis in response to a client's request for a segment. Consequently, in this thesis,

we are interested in seeking solutions to these two related but distinct objectives:

- **Objective 1 (O1)**: Design a practical distributed scheme for real-time transcoding.
- **Objective 2 (O2)**: Design an adaptive algorithm optimized for live transcoding and high QoE.

Based on these objectives, we further describe a set of questions and challenges related to each objective.

1.3.1 Objective 1: Design a practical distributed scheme for real-time transcoding

As stated previously, the transcoding rate must at least be as high as the live video rate to sustain a live stream. If this condition is not met, then multiple segments must be transcoded concurrently to make the effective transcoding rate meet the live rate. Therefore, we outline the following questions related to real-time transcoding:

- **O1.Q1** How to tune the transcoder to get real-time performance?
  We explore the parallelization options offered by the transcoder to speed up the transcoding process.
- **O1.Q2** How to make the transcoding process scalable and resource efficient?
  We use lightweight Docker containers to make the transcoding process resource-efficient and scalable.
- **O1.Q3** How to estimate the resources needed to reach a target transcoding throughput?
  We propose Machine learning based solutions to estimate the transcoding time and resource requirements for achieving real-time performance.

1.3.2 Objective 2: Design an adaptive algorithm optimized for live transcoding and high QoE

With on-demand transcoding, it is possible to dynamically adapt to the client's needs and deliver the most optimal quality segment to the client in real-time to maximize QoE. Therefore we outline the following questions related to high QoE live video transcoding and streaming:

- **O2.Q1** Given the competitive nature of QoE metrics (quality, stalls and delay), how to optimize the overall QoE?

  We explore a loss function which relates the QoE metrics and the overall QoE can be maximized based on the loss function.

- **O2.Q2** How to customize the transcoding server and the video player for adaptive high quality streaming?

  We design and implement `Antifreeze` which is an end-to-end on-demand transcoding, and streaming solution for delivering high quality streams in real-time.

- **O2.Q3** How does the proposed system perform under different streaming environments?

  We evaluated the performance of `Antifreeze` for different kinds of video content under different bandwidth profiles.

1.4 Research Contribution

Existing solutions to enhance the video streaming pipeline can be broadly classified into two categories – Transcoding solutions and adaptive streaming solutions. Transcoding solutions focus on how to effectively transcode video segments. This includes speeding up the transcoding/encoding speed [30, 31, 32, 33], using distributed computing to parallelize the transcoding job across multiple processing nodes [34, 35, 36, 37] (suitable for stored videos and not live videos) or using containers to easily deploy the transcoding services on any

infrastructure [38]. Adaptive streaming solutions on the other hand focus on maximizing the viewer's Quality of Experience. This includes ABR rules such as throughput rules, buffer rules [39, 40], visual quality rules, and other rules. However, to make the end-to-end live system more efficient, transcoding and adaptation should be coupled under one umbrella as both are essential components of the live streaming system. In this thesis we try to fill this gap in literature by coupling video transcoding and video quality adaptation stages. By coupling video transcoding and quality adaptation, it is possible to deliver the best fit quality segment to the client which can maximize the client's QoE.

For our research, we used the High Efficiency Video Coding (HEVC) standard as our encoding tool since it has a high compression efficiency and thus enables high-quality streaming under limited bandwidth. To achieve real-time transcoding, we first evaluated the tools offered by the codec to speed up the transcoding process (O1.Q1). Our results indicate that depending upon the complexity of the transcoding task, the transcoder throughput may not always be sufficient to sustain a live stream, and there may be a need for distributed concurrent segment transcoding. To facilitate distributed transcoding, we explored the use of Docker containers as lightweight transcoders that can be deployed easily on the cloud, thereby leveraging a large amount of computational resources in a scalable manner. Furthermore, transcoding is a highly resource-intensive process, and with containerized transcoders, it is possible to balance the trade-off between transcoding throughput and the resources consumed (*e.g.,* cores) as the resources allocated to a container can be controlled in a fine-grained manner [41] (O1.Q2).

From literature review and our own empirical analysis, we concluded that for a transcoding task, the transcoder throughput and the resources needed to achieve a transcoding throughput correlate with the segment properties and the transcoding parameters. We leverage this relationship to determine the minimum size of the container (*i.e.,* cores) needed to achieve a target transcoding rate (O1.Q2). We propose machine learning (ML) based models to make per-segment transcoding predictions (O1.Q3). Specifically, we crafted a transcoding

dataset and implemented two machine-learning models. The first model is a regression model and predicts the transcoding time given the segment properties, the transcoding parameters, and the transcoding resources (*i.e.,* cores). The second model is a classification model and predicts the minimum number of cores required to reach a target transcoding rate.

The regression model had a Mean Absolute Error of 0.116 seconds and a Mean absolute percentage error of 2.191% in predicting the transcoding time. The classification model had an accuracy of 98.3% and an f-1 score of 0.97 in predicting the CPU cores required to reach a target transcoding throughput. Additionally, the models have significantly fewer overheads, and thus integrating the models in the transcoding pipeline does not introduce any significant delays. When used in conjunction, the two models can help balance the trade-off between transcoding rate and CPU cores in the context of real-time transcoding. Our results indicate that using the prediction model to provision resources leads to resource savings compared to a fixed resource allocation strategy that does not consider the per-segment transcoding performance when allocating resources.

Apart from seeking solutions to direct the real-time transcoder to transcode videos at a rate matching the live video rate, it is equally important to make the transcoder adaptive to the client's needs given the client's playback conditions (*i.e.,* bandwidth and buffer) may change over time. To deliver a high-quality, smooth stream to the end user in real-time, we propose `Antifreeze`: an end-to-end adaptive platform for real-time transcoding and streaming. To begin with, we first studied the client buffer behavior for a real-time transcoded live stream. In particular, we evaluated the buffer change over the period of time a live segment is transcoded and delivered to the client. Then based on the buffer model, we proposed a QoE loss function designed to keep the buffer in a healthy state (to minimize the chances of stalls) and to deliver high and consistent quality video segments to the end users (O2.Q1). We then designed and implemented `Antifreeze`: a real-time transcoding and streaming platform to provide a more personalized live streaming experience to the end-user (O2.Q2). Specifically for a real-time transcoding task $s \rightarrow s'$, where s is the source

segment and s' is the transcoded segment, antifreeze computes the most optimal s' whose quality is close to the user requested quality and which can be generated (via transcoding) and delivered to the client at a rate fast enough so that the client buffer does not drop to critical levels during the process.

We deployed `Antifreeze` under different bandwidth profiles and evaluated the performance of `Antifreeze` by examining the visual quality of the stream and the stall events. Results show that `Antifreeze` can deliver a high-quality stream in real-time with minimum playback stalls under various bandwidth profiles (O2.Q3). Specifically, we investigated the impact of `Antifreeze` tuning parameters on the stream performance. We then compared antifreeze with other baseline on-the-fly transcoding algorithms and concluded that antifreeze resulted in fewer playback stalls compared to the other algorithms.

To summarize, the contributions of this thesis are twofold. Firstly, we developed two prediction models, one to predict the transcoding time and the other to predict the number of CPU cores needed to reach a target transcoding time. The two models can help optimize the streaming service. When the streaming session begins the goal is to fill up the buffer quickly so that playback can start quickly. Under such a scenario, the transcoding time prediction model can guide the transcoder to transcode the stream at the fastest rate. Once playback begins and the player enters a steady state, the objective should be to keep the buffer level constant by making the content delivery rate match the video playback rate. Under such a scenario, the CPU cores prediction model can guide the resource allocation to the transcoder. Secondly, we implemented antifreeze which is a real-time adaptive streaming platform. Antifreeze adapts to the client needs during run-time to deliver the most optimal quality segments to the client which maximizes the clients QoE.

1.5 Thesis Organization

This thesis has been organized into eight chapters. A brief overview of each chapter is as follows:

- Chapter 1 presents an overview of a live streaming setup along with the associated challenges. Primarily, transcoding and QoE related challenges are outlined. Furthermore research objectives of the thesis are discussed and finally the main contributions of the work are summarized.

- Chapter 2 presents the essential video streaming background by providing an overview of video transcoding and video streaming over HTTP. Docker based containerization is also discussed.

- Chapter 3 provides overview of related work in the field of transcoding and adaptive live streaming.

- Chapter 4 provides an overview of the HEVC video coding standard. In particular the tools offered by the codec to speed up the transcoding process are studied. Finally, containerization of the codec using Docker is described.

- Chapter 5 presents the machine learning models employed to predict the transcoding time and the transcoding resources. Specifically, dataset generation, dataset analysis and ML model generation is detailed out. Furthermore the model's performance is evaluated and the usefulness of the model in the context of live streaming is highlighted.

- Chapter 6 presents the design of an on-demand real-time transcoding platform - `Antifreeze`. The client buffer model for an on-demand real-time stream is first derived. Adaptive algorithm to maximize the client QoE is then described.

- Chapter 7 presents the implementation details of `Antifreeze`. The performance of antifreeze under different bandwidth profiles is evaluated. Comparison of `Antifreeze`

with baseline techniques is also presented.

- Chapter 8 concludes the thesis by proving a brief outline of the work carried out. It also discusses the limitations of this the work and relevant future work.

Chapter 2

Background

In this chapter we present the background information related to the thesis. Firstly, we review the fundamentals of video transcoding in Sec. 2.1. Next, we discuss about video complexity and quality assessment in Sec. 2.2. Then we discuss about HTTP adaptive Streaming protocols used to deliver videos to the client in Sec. 2.3. Finally, in Sec. 2.4 we discuss about Docker.

2.1 Video Transcoding Fundamentals

Video transcoding consists of changing one or more properties of a video. It involves taking an encoded video, decoding it, and then encoding it again with the desired properties [42, 21, 22]. Common transcoding tasks include changing the video encoding format (codec), changing the bitrate of the video (transrating), or changing the resolution of the video (transizing) [42]. Given the heterogeneity of viewing devices and network connectivity, video transcoding ensures that the media content is suitable for consumption by diverse end users thus enhancing the end user's Quality of Experience (QoE).

A video is a sequence of frames played at a rate fast enough to give a notion of continuous movement. Frames per second (fps) is the number of frames constituting one second of video playback. Each frame in a video is a 2-dimensional (rectangular) array of pixels where the

array dimensions (width × height) define the resolution of the video. The number of bits needed to store a pixel depends on the color-space model used and is 24 bits for RGB 8 color space and 12 bits for YUV 420 color space [43]. Regardless of the color space used to represent a pixel, the size of a raw video file is too large to be stored or to be transmitted over the internet. Therefore, video files are compressed using a video codec like HEVC which describes how a raw video is compressed and decompressed. A codec reduces the file size of the video while preserving its visual quality. It does so by looking for redundancies within a video frame (spatial compression) and across video frames (temporal compression) which results in compression. Similarities exist between adjacent frames in a video and instead of storing similar information across video frames redundantly, a codec encodes the difference it observes between adjacent frames (motion compensation [44]) resulting in bit savings.

From an encoding perspective, a video is segmented into Group of Pictures or GOP's where each GOP is a short sequential arrangement of frames. Depending on temporal (or inter-frame) compression, a GOP frame can be one of the three types – I frame (Intra frame) or a P frame (Predicted frame) or a B frame (Bi-directional frame) [45, 46, 47]. Temporal compression does not exist for I frames meaning that I frame does not depend on any other frame whereas temporal compression exists for P frames and B frames. Specifically, P frames can use preceding frames as temporal references, and B frames can use preceding as well as succeeding frames as temporal references. The order of frame types in increasing order of compression is I frame, P frame, and finally B frame [45]. Temporal compression results in dependencies among the video frames in a GOP and Fig. 2.1 shows the video frames within a GOP along with the temporal dependencies.

The Bitrate of a video represents the number of bits required to encode each second of that video and is often computed in kilobits per second(kbps) or megabits per second(mbps). The bitrate of the video depends on the video properties (content complexity, resolution, fps) and the codec used. Bitrate is a crucial factor when streaming videos over the internet. If the video bitrate is higher than the network bandwidth, it will result in the rate at which new

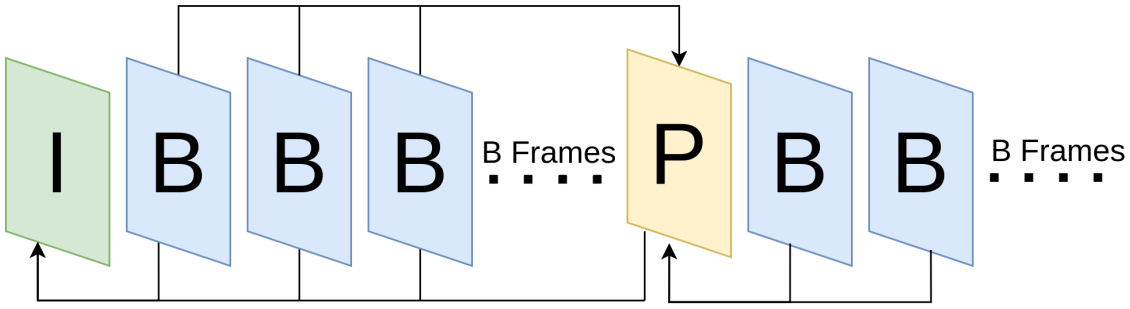


Figure 2.1: GOP structure with I, P and B frames and their dependencies

content is being downloaded by the client to lack the rate at which the client buffer is being emptied out due to playback leading to stalls. Hence encoding a video with a minimal number of bits (high compression efficiency) while still retaining good visual quality is desirable. For Videos in the 4K range and beyond (360-degree videos) keeping a check on the bitrate is an even bigger challenge due to the sheer volume of data contained in them. Traditional video codecs like Advanced Video Codec (AVC) fail to provide the necessary compression efficiency needed to stream high-resolution/high-quality videos over conventional networks. High Efficiency Video codec popularly known as HEVC is a next-generation codec that has superior coding efficiency compared to AVC and can encode videos at the same visual quality but with a less number of bits [48, 49] and is, therefore, an ideal codec to encode high resolution videos. Because of this benefit we adapted HEVC as the transcoding codec for our research and it is described in detail in Sec. 4.1

2.2 Video Complexity and Quality Assessment

Videos differ in genre and various kinds of video content have different compression complexities. Animated videos have slow-moving objects and flat regions (similar pixel regions) which can be encoded with a fairly less number of bits [50] whereas action videos have fast-moving objects which makes temporal compression difficult resulting in higher processing time as well as higher encoding bits. Spatial Information (SI) and Temporal Information

(TI) are key metrics to measure the spatial and temporal complexity of videos, respectively [51, 23].

According to ITU-T, Spatial Information (SI) is a measure of information (details) contained in an image [51]. To compute the SI value each frame of a video is filtered with Sobel filter - an image processing technique that detects edges in an image [52]. Edges are pixel regions that are dissimilar to their neighbors and edges are a good indicator of the details contained in a frame. For each of the Sobel-filtered frames, the standard deviation across the entire pixel space is calculated. The highest value of standard deviation over all the video's frames is then the final SI value for the video as shown in Equation 2.1 [51].

$$SI = max\{\sigma[Sobel(F_n)]\} \tag{2.1}$$

Where F_n is the n^{th} frame of the video.

According to ITU-T Temporal Information (TI) is a measure of the amount of motion in a video and is higher for fast-moving videos [51]. To compute the TI value, the pixel difference between pixels located at similar 2-d coordinate location in adjacent frames is calculated. This difference indicates how much a pixel change over time and the maximum value of standard deviation across all of the computed frames is then the final TI value of the video as shown in Equation 2.2 [51, 23].

$$TI = max\{\sigma[F_n(i,j) - F_{n-1}(i,j)]\} \tag{2.2}$$

Where F_n and F_{n-1} are the n^{th} and $n-1^{th}$ frame of the video respectively and i,j is the 2-dimensional coordinate location of the pixels in those frames.

While SI and TI values can help determine the compression efficiency and the video bitrate, they do not directly translate to how a video will be qualitatively perceived by the end user. There are many metrics to assess the perceptual appeal of a video. These include Peak signal to Noise Ratio (PSNR), Structural similarity Index (SSIM) and Video

Multi-method Assessment Fusion (VMAF). PSNR is a measure of pixel degradation that has occurred because of the lossy video compression [53]. It has a typical range of 30 – 45 dB [54] with higher values indicating less pixel degradation and better visual quality. SSIM is based on the similarity between an image and a reference (uncompressed) image [53] and has a range of 0 to 1 with a higher value indicating more similarity and better visual quality. PSNR and SSIM are objective metrics and do not consider how humans comprehend videos. VMAF [55, 56] on the other hand is a relatively new subjective video quality metric developed by Netflix. VMAF superimposes machine learning techniques to human vision [56, 53] to better understand how humans comprehend a video and thus provides a better understanding of perceived video quality compared to other metrics [57, 58]. VMAF value ranges from 0 to 100 with higher VMAF values implying higher visual qualities. For our research, we relied on VMAF as the visual quality indicator.

2.3 HTTP Adaptive Streaming

RTMP [59] was traditionally considered the prominent protocol to deliver media streams online. It has now been replaced by HTTP Adaptive Streaming commonly referred to as HAS. HTTP Live Streaming or HLS [60] is one of the earliest HAS standards and was released by Apple as a proprietary standard back in 2009. As opposed to HLS, Dynamic Adaptive Streaming over HTTTP or DASH [61] is an open-source HAS standard developed by the Moving Picture Experts Group (MPEG) and published in 2012. DASH has been widely popular in the streaming industry and applications such as YouTube, Netflix, and Hulu have adopted DASH [62]. DASH can deliver live streams as well as video on demand (VoD) streams over the internet to a variety of devices ranging from wide 4K television to small mobile devices under different network connectivity. Additionally, DASH is codec agnostic and can work with any compression standard.

DASH is designed to provide a smooth video playback experience to clients even under

Table 2.1: Bitrate Ladder adapted by Netflix [13]

Bitrate (kbps)	**Resolution**	**Bitrate (kbps)**	**Resolution**
235	320x240	375	384x288
560	512x384	750	512x384
1050	640x480	1750	720x480
2350	1280x720	3000	1280x720
4300	1920x1080	5800	1920x1080

unstable or fluctuating network conditions. With DASH, a video is segmented into smaller segments (also known as chunks in some literature) and each segment is transcoded at multiple quality levels or representations. Each representation is characterized by a bitrate-resolution pair and the set of representations collectively forms a bitrate ladder. The bitrate ladder is designed to cover the widest possible range of quality levels thus enabling adaptive video streaming at high quality (when there is sufficient bandwidth/buffer) as well as low quality (when bandwidth/buffer is low). Table 2.1 depicts the bitrate ladder employed by Netflix to encode and package its video contents. The highest representation offered is 1920X1080 @ 5800 kbps and the lowest one is 320X240 @ 235 kbps. Each DASH segment is a few seconds long and the DASH clients can dynamically change the quality level between segments to get smooth playback free from stalls. Fig. 2.2 illustrates a DASH streaming session with 3 representations and the DASH client switches representations between segments to get a smooth playback at the highest possible quality level.

When a DASH server transcodes a video to multiple representations, it creates a stream file known as Media Presentation Description (MPD) which contains data about the segmented video (video chunks) including the bitrate ladder (available representations), segment duration, download URLs, etc. To commence a DASH streaming session, the player first requests the MPD file from the server. It parses and decodes the MPD file and starts downloading the segments into its buffer (player memory) in sequential order. When sufficient segments have been downloaded and the buffer level reaches a threshold value, the player starts pushing out the segments from the buffer onto the screen and thus playback begins. Although a buffer results in playback delays as the player must temporarily store a

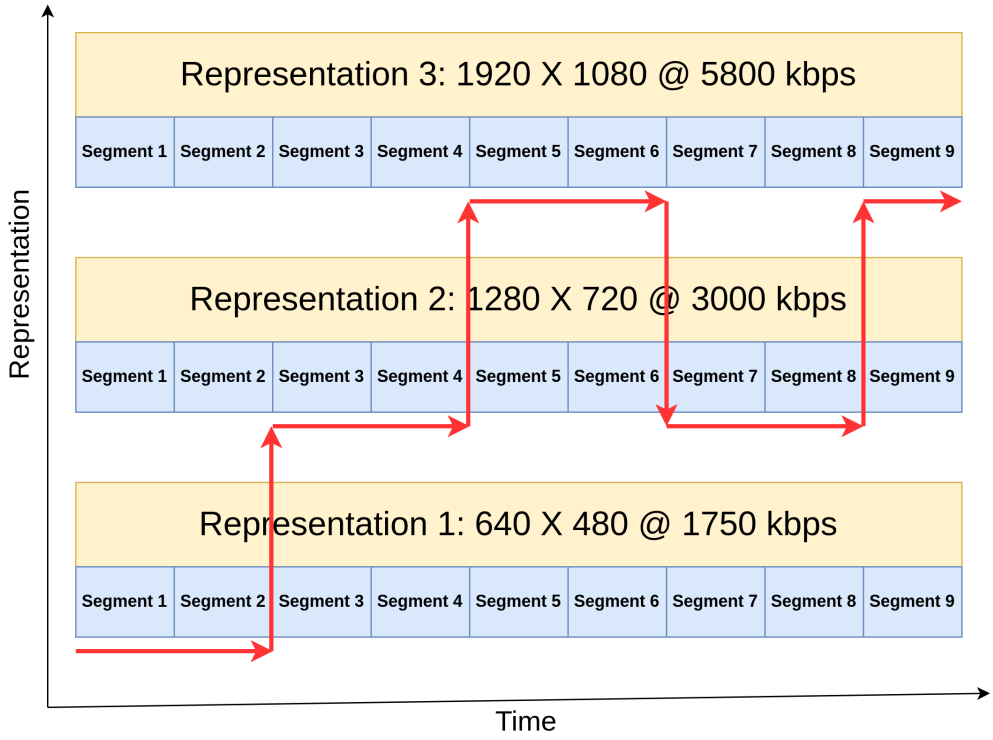


Figure 2.2: An example of a DASH streaming session with 3 representations [2]

few segments before beginning playback, having a buffer ensures that the player can absorb the effects of sudden bandwidth drop which results in the rate at which new streamed content is being downloaded to lag the rate at which buffer is being emptied out due to playback. During the streaming session, the player continuously monitors the bandwidth and buffer occupancy and queries a set of rules known as adaptive bitrate (ABR) rules to determine the quality level of the next segment that it should request from the server. This adaptive quality switching during playback ensures continuous playback without stalls and optimal utilization of the available bandwidth.

Primary ABR rules used to determine which quality level segment to request next from the server are either throughput-based, buffer-based, or a combination of both (hybrid rules) [39, 40]. Every time the player fetches a segment from the server it records the bandwidth/throughput between the server and the player. Throughput rule then select the quality level whose bitrate is below the estimated bandwidth. This ensures that the player buffer

never drains out. The estimated bandwidth is often scaled down to make the throughput rule more conservative. For instance, `dashjs` an open-source DASH client scales down the estimated bandwidth by a factor of 0.9 to prevent buffer stall risks [40]. Buffer rules rely on the level of buffer occupancy to determine the next quality level [40]. Buffer rule generally selects bitrate level (& thereby quality) proportional to the level of buffer occupancy. i.e. they select higher bitrates (& thereby quality) when the buffer level is high and vice versa. Buffer-based rules work best for VOD streams that have a large buffer and are suboptimal for live streams where the player must operate with a small buffer to minimize the delay. Hybrid rules use a combination of throughput rules and buffer rules to make a more informed track selection decision. Apart from the primary ABR rules there are many other secondary ABR rules such as Insufficient Buffer Rule, Switch History Rule, Dropped Frames Rule etc. which are designed to make quality selection decisions for specific scenarios [39, 40]. Switch History Rule, for instance, does not permit extreme quality level switches between segments to ensure that the stream has a consistent video quality.

2.4 Overview of Docker

Virtualization was introduced in the late 1960's by IBM to increase the resource utilization of a computing system [63]. Today there are two popular virtualization techniques - hypervisor/Virtual machine (VM) based virtualization and container-based virtualization. Hypervisor is a tool that creates and manages virtual machines over the host system. Two of the popular hypervisor-based virtualization tools are `VMware` and `VirtualBox`. Virtual machines are heavyweights as they need a dedicated OS and consume a large share of the host system's memory and computing resources. Docker [64] on the other hand is a lightweight container-based virtualization tool developed by Solomon Hykes and Sebastian Pahl and released in 2013. Docker containers are lightweight and the performance of containers is similar to native implementations [65]. Docker containers do not need a dedicated os like

VM's, instead they share the host OS [66]. The launch time of a Docker container is significantly less than that of VMs [67, 68]. Additionally, the memory and disk usage of container is also much less than that of a VM which allows multiple containers to run in parallel on a single host [68].

For our research, we used docker containers as lightweight, portable, and standalone transcoders that can be deployed on the cloud to execute video processing tasks. To create a containerized transcoder the first requirement is to create a docker image that contains all the required video processing dependencies. We configured the image with `FFmpeg` [69], a popular open-source video processing library. From the transcoder image, containers can be launched within a fraction of a second which then acts as independent transcoders having all the necessary video processing libraries built in. A detailed description of our containerized transcoder is presented in Sec. 4.3

Chapter 3

Related Work

This chapter reviews existing works on video streaming along two dimensions. Firstly, in Sec. 3.1, we review methodologies to characterize the transcoding performance (*i.e.,* transcoding time and resource allocation) as it is often the transcoding stage, which decides if a live video can be streamed in real-time. Secondly, in Sec. 3.2, we provide an overview of adaptive streaming solutions, including on-the-fly transcoding and quality-aware streaming designed to offer a high QoE to the viewers.

3.1 Transcoding Performance

In this section, we first explore existing methodologies to predict a video segment's transcoding time (Sec. 3.1.1), which helps us gain insights into the factors that impact the transcoding time. Then we provide an overview of containerized transcoding solutions (Sec. 3.1.2) which help in the easy deployment of transcoding services and also help to allocate resources efficiently.

3.1.1 Transcoding Time Prediction

Many studies to predict the transcoding time of video segments have been proposed in literature over the years. Deneke et al. implemented three types of machine learning models to predict the transcoding time of segments - Linear Regression (LR) model, Support Vector Regression model, and Neural Network model [70, 71]. Their results suggested that out of the three models, NN has the best performance in terms of accuracy. The transcoding parameters they considered when training the models included bitrate, resolution, frames per second (framerate), and the video codec. Li et al. observed that GOPs within a video are correlated and based on this correlation they further implemented a model for estimating the transcoding time of a GOP [72]. In another work, Li et al. implemented a regression model to evaluate the impact of GOP size and frame rates on transcoding time and concluded that the number of frames in a GOP has a good relationship with the transcoding time [73]. A high correlation between GOP size and transcoding time was also confirmed by Krishnappa et al. when they reported a linear relation between transcoding time and segment duration [74]. Ma et al. proposed a statistical and probability-based transcoding time estimation method taking into account the segment duration and target bitrates (quality levels) [75].

Aside from predictions based on GOP/segment properties, other models that also consider video complexity when making predictions have also been proposed. Zabrovskiy et al. included the segment complexity - Spatial Information (SI) and Temporal Information (TI) as a feature to train their ANN model [23]. The authors showed that the inclusion of segment complexity leads to more accurate transcoding time predictions. They also proposed a fast approach to measure the complexity of a segment by transcoding the segment to a low-resolution variant and computing the SI and TI values for the low-resolution variant. There exists a good correlation between SI and TI values across resolutions [76]. SI is a measure of information density, and with downscaling the same information is confined to a much smaller area leading to an increase in SI value [76]. Conversely, TI is a measure of motion present in a video sequence and does not depend on the video resolution [76]. In

another work, Zabrovskiy et al. downscaled the segments to a 256x144 resolution and obtained the 144p transcoding time and the 144p file size as complexity indicators to improve the accuracy of their ANN model [77]. Zhao et al. came up with a model to estimate the complexity of a video transcoding task [78].

In addition to the segment properties and content complexities, the computing resources where the transcoding is carried out also determine the transcoding time. Li *et al.* [73] investigated the transcoding performance across different VM instance types, namely General Purpose, CPU optimized, Memory Optimized, and GPU, and offered a comparison across the instances. They showed that the motion present in a video plays a vital role in determining the transcoding performance on a VM. For low-motion videos, GPU instances outperform the other instance types. Additionally, the transcoding time also depends on the amount of resources allocated to the transcoder. Sameti *et al.* [38] reported that the number of cores allocated to the transcoder plays an important role in determining the transcoding speed.

3.1.2 Containerized Transcoding

Deploying transcoding services to the cloud becomes easy with containers, and several such container-based transcoding solutions have been proposed in prior works [79, 80, 81, 82, 83, 84, 38]. Paakkonen *et al.* predicted the transcoding performance (*i.e.,* transcoding time and CPU usage) for transcodings carried out on docker containers deployed on Virtual Machine (VM) instances of several sizes (*i.e.,* 4, 6, 8, 16 vCPUs) [79]. Their system recommends a suitable transcoding resource based on a transcoding need. Extending their work further, Paakkonen *et al.* implemented models to predict the video transcoding performance (transcoding time and CPU usage) on a Kubernetes system [80]. Sameti *et al.* proposed CONTRAST, a distributed container-based transcoding framework that exploits a profiling technique to estimate the number of cores required to sustain the frame rate according to the resources available in the hosting cloud [38]. They showed that the number of cores needed for a transcoding task is a function of source and target resolutions.

Table 3.1 provides a comparison between previously proposed transcoding time prediction models and the model presented in this thesis. Although prior proposed models [23] have a high accuracy in predicting the transcoding time, the authors performed transcoding using a single-threaded mode which is too slow to be employed for real-time transcoding. Furthermore, to the best of our knowledge, our research is the first one that considers the cores allocated to a transcoder as a feature to train the transcoding time prediction model.

Table 3.1: Proposed Transcoding Time Prediction Models

Id	Prediction Model	Codecs/encoders	HEVC	Ultra HD	Comments
[70, 71]	Artificial Neural Network (ANN), Support Vector Regression (SVR), Linear Regression (LR)	AVC, Mpeg4 Vp8, H263	No	No	Included bitrate, resolution, fps, and video codec type as transcoding features
[72]	Linear	AVC	No	-	Transcoding time is function of GOP size Sn and average GOP size Savg
[23]	Artificial Neural Network (ANN)	AVC, HEVC	Yes	Yes	Included segment complexity (Spatial Information and Temporal Information) for the prediction model
[77]	Artificial Neural Network (ANN)	AVC	No	Yes	Included 144p transcoded video properties (Bitrate and Transcoding Time) for the prediction model
[79]	Random Forest	AVC	No	No	Used a Docker based system to perform the transcoding
[80]	Random Forest Regressor (RFG) Reinforcement Learning (RL), Stochastic Gradient Descend (SGD)	AVC	No	No	Used a Kubernetes based system to perform the transcoding
[73]	Second Order Regression Model	MPEG2, AVC, HEVC	Yes	-	They used different VM categories (General-Purpose, CPU-Optimized, Memory-Optimized and GPU VM)
[75]	Statistics and Probabilistic Theory	MPEG-4	-	No	-
Our	Google Vertex AI	HEVC	Yes	Yes	Included CPU cores allocated to transcoder (docker container) for the prediction model

3.2 Adaptive Streaming

The streaming adaptations which ensure a smooth streaming experience can be broadly classified into stream switching adaptations, and on-the-fly transcoding adaptations [85]. Most traditional HAS services (Sec. 2.3) are based on stream switching, where the server transcodes a video stream into multiple quality levels and the client picks the appropriate quality level according to the observed bandwidth, buffer level and other playback conditions. On the other hand, transcoding based adaptation is performed on-the-fly by the server in real-time during the streaming session. For transcoding adaptation, there are on-the-fly transcoding solutions proposed in literature [25, 26]. Unlike conventional stream switching based on a fixed set of quality levels, with real-time transcoding, it is possible to dynamically adapt to the client's needs and playback conditions during run-time, thus offering a more personalized streaming experience to the viewer. On-the-fly transcoding could potentially lead to a better quality stream and bandwidth utilization. Additionally, for live streams where the content needs to be delivered as soon as it becomes available (it is generated), the transcoding has to be performed on-the-fly in real-time. In this section, we first outline some of the proposed on-the-fly transcoding solutions followed by an overview of QoE aware streaming schemes.

3.2.1 On-the-fly Transcoding Solutions

On-the-fly transcoding does not only help in enhancing the end user's QoE but also helps in reducing the overall transcoding and storage costs. Zhao *et al.* suggested that popular video segments should be pre-transcoded at several quality levels and stored, but less popular video segments should be transcoded on-the-fly [25]. In their setup, a requested segment is either directly delivered if it is stored or otherwise transcoded on-the-fly and delivered. They optimized the trade-off between storage cost and on-the-fly transcoding cost to determine whether a segment should be stored in advance or generated on-the-fly by

means of on-demand transcoding. Darwich *et al.*. also proposed a method to minimize the operational cost by balancing the trade-off between pre-transcoding and on-the-fly transcoding [26]. Based on the cost incurred, their algorithm decides whether a video should be pre-transcoded and stored or should be transcoded on-the-fly. Erfanian *et al.* proposed a strategy whereby only the highest quality segment is transmitted to the edge server, and the edge server dynamically transcodes to other quality variants to minimize bandwidth usage and operational costs [86]. To speed up the transcoding process for live streams, the edge server utilizes the encoding metadata from the previous transcodes. Krishnappa *et al.* proposed a model to transcode a segment on-the-fly to bitrates that the client might request next [74]. The authors used Markov chain to predict the possible bitrate the client might request in the future [74]. In this thesis, we take transcoding adaptation (on-the-fly transcoding) to the next level. We propose an algorithm (Chapter 6) to pick the transcoding parameter to produce a transcoded segment with the highest possible visual quality but small enough to meet the end-to-end delay constraint (including transcoding delay and streaming delay).

3.2.2 QOE Aware Streaming Schemes

While Throughput-based and Buffer-based adaptations continue to be the primary ABR algorithms [39, 40], algorithms that also take into account the visual quality of the stream have also been proposed. Qin *et al.* [16] in their research presented the viewers with three quality levels: good, better, and best. The quality level selected by the user was then mapped to a VMAF value. To enhance the QoE they proposed Chunk Based Filtering (CBF) and Quality Aware Data Efficient Streaming (QUAD) strategies. CBF discards those quality levels from the bitrate ladder whose VMAF value exceeds the user requested VMAF value and this filtering results in bandwidth savings. QUAD is an ABR scheme that not only considers the bandwidth but also the video quality and video quality switches when making track selection decisions. QUAD leads to a consistent quality stream that matches the user requested target quality and suffers fewer stalls. The authors of [27] argued that viewers

prefer a slow continuous switch between extreme quality levels (instead of a sudden switch) and designed QDASH-qoe a client-side adaptation logic that picks up intermediate quality levels rather than abruptly switching between extreme quality levels. Proportional integral derivative(PID) controller based schemes like PIA [17] and CAVA [28] which relies on a control function to maximize video quality and minimize quality switches and stalls have also been proposed in literature. To minimize quality fluctuations in the stream Li *et al.* conveyed the quality information of segments to the clients and implemented PANDA [29] a dynamic programming solution to make a more informed track selection decision.

Dynamically tuning the ABR rules based on the playback conditions to maximize the QoE has also been proposed in prior works. Mao *et al.* proposed Pensieve [87], a scheme which instead of using a predefined static ABR rules, uses machine learning to dynamically determine the ABR rules based on clients playback history leading to an enhancement in QOE. Similarly, Akhtar *et al.* designed OBOE [88] which tunes the parameters of an ABR algorithm at run time based on the networking conditions resulting in an enhanced QOE.

Enhancing the QoE for live streams has also been proposed in literature. Lan *et al.* modelled the client buffer for a live stream and implemented a buffer-based ABR scheme for low latency live streaming and showed that their proposed scheme reduces quality switches in a live stream [89].

While many quality-aware ABR algorithms have been proposed in literature, most of them have been designed for VOD applications with a focus on stream switching adaptations and not on-the-fly transcodings adaptations. Considering the transcoding time together with bandwidth, buffer, and video quality while making on-the-fly transcoding choices for a live stream is the major research focus of this thesis.

3.3 Summary

In this chapter, we first reviewed methodologies to predict the transcoding time and the transcoding resources. From these studies, we conclude that the transcoding time depends on the segment/GOP properties, the transcoder parameters, and the underlying transcoding infrastructure. We leverage this relationship to come up with a solution to transcode a video stream in real-time while just using the right amount computational resources in Chapter 5

We then reviewed adaptive streaming solutions which can enhance the viewers QoE. In particular, we reviewed on-the-fly transcoding solutions that can help enhance the client's QoE as well as save operational costs for the stream provider. Following this, we outlined QoE Aware adaptation algorithms which consider the video quality when making track selection choices to improve the QoE.

While many quality aware ABR algorithms have been proposed in literature, most of them have been designed for VoD applications with a focus on stream switching adaptations and not on-the-fly transcoding adaptations. Considering the transcoding time together with bandwidth, buffer, and video quality while making on-the-fly transcoding choices for a live stream is the major research objective of this thesis and in Chapter 6, we design and implement such a real-time transcoding solution designed for live-streams.

Chapter 4

Benchmarking distributed transcoding

In this chapter, we first provide an overview of the HEVC coding standard in Sec. 4.1. We then analyze the codec by studying the relationship between the codec parameters and the transcoding performance in Sec. 4.2. Finally, in Sec. 4.3 we describe containerized transcoding.

Increasing demand for live and interactive streaming applications calls for effective video compression without a degradation in the video quality. This challenge can be met by the High-Efficiency Video Coding (HEVC) standard, which despite being a computationally complex standard [90], can encode videos at a much lower bitrate compared to legacy codecs making it an ideal codec for high-quality streaming under low bandwidths.

4.1 Overview of HEVC

HEVC was standardized by ISO/IEC and ITU-T and published in 2013 [12]. Studies show that HEVC has superior coding efficiency compared to traditional coding standards like Advanced Video Codec (AVC) and can encode videos at the same level of visual quality but with a smaller number of bits [48, 49]. Similar to other codecs, HEVC looks for redundancies in a video to compress a video. HEVC encoder divides a video frame into blocks called Coding Tree Units or CTUs whose sizes can be 16x16, 32x32 or 64x64 pixels [12, 5, 91, 3].

The larger CTU blocks are further subdivided into smaller square partitions called Coding Units or CU and the CU's can further be recursively subdivided into four smaller CU's as shown in Fig. 4.1. The highlighted CTU (green block) in Fig. 4.1 is split into four 32x32 CU's and the top right highlighted CU (blue block) is further divided into prediction units or PU and the PU's are encoded based on spatial(intra-frame) or temporal (inter-frame) similarities.

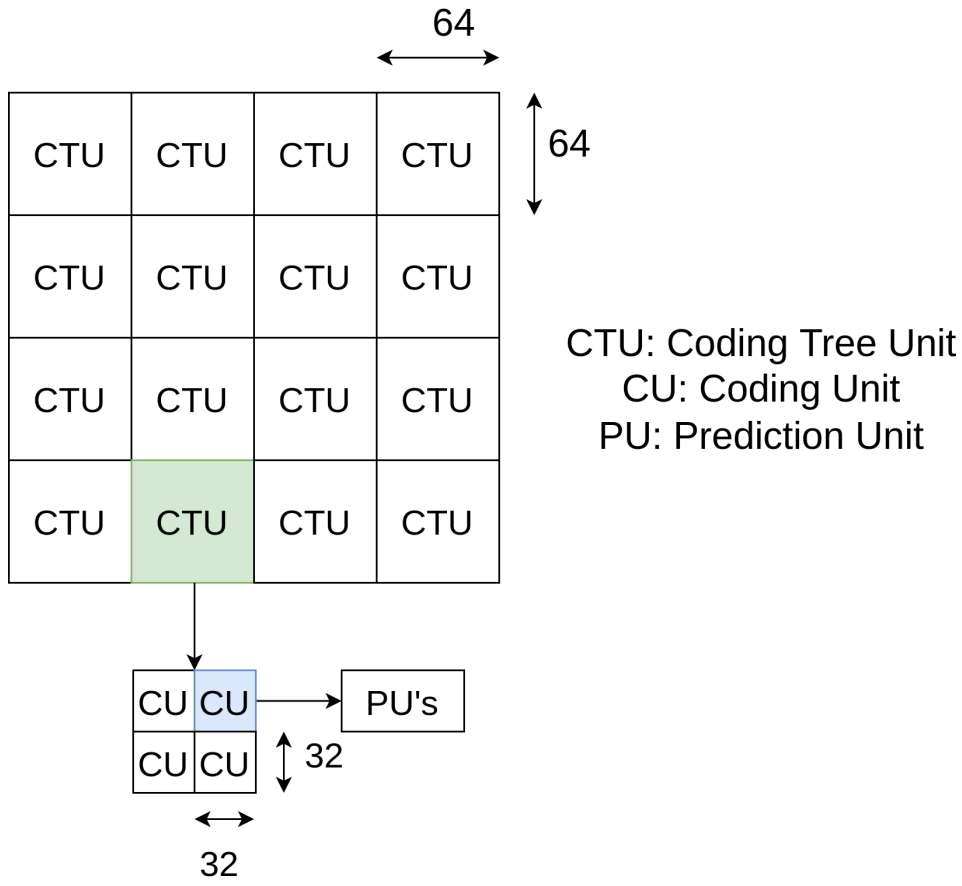


Figure 4.1: Frame partition in HEVC into quadtree blocks [3, 4, 5, 6]

Spatial and temporal prediction are the two prediction modes used to predict the value of a pixel based on a reference pixel(s) value. Spatial prediction is an intra-frame prediction and HEVC supports 35 spatial prediction modes to predict a PU block from reference blocks located within the same frame [91]. Temporal prediction is an inter-frame prediction that considers the movement of objects across video frames (motion compensation vectors) to

predict a PU block. Spatial prediction introduces CTU block dependencies within a frame and temporal prediction introduces block dependencies among frames within a GOP. Based on temporal dependencies, frames can be classified into reference frames and non-reference frames. Frames that rely on other frames for motion estimation are known as non-reference frames or dependent frames and the frames on which they rely are known as reference frames.

The highly optimized encoding techniques employed by HEVC result in improved compression efficiency but make the codec slow and computationally demanding. To overcome this limitation, HEVC is designed to encode multiple CTU blocks concurrently and offers parallelism tools namely Inter Frame Parallelism, Wavefront Parallel Processing, and slices to speed up the transcoding process. These parallel processing schemes are summarized below:

- *Inter frame parallelism:* Inter frame parallelism [92] is the concurrent processing of multiple frames within a GOP. `libx265` [92] a popular HEVC encoder uses frame threads to encode frames concurrently. Using a single frame thread requires much fewer CPU resources as a single frame gets processed at a time but has higher encoding times. However, over-allocating frame threads will also not yield any performance benefits [92] as temporal dependencies exist among frames and non-reference frames must wait for the reference frames to be encoded before motion predictions can be made.

- *Wavefront Parallel Processing:* WPP [93] is a Spatial or intra-frame parallelism mode whereby multiple CTU rows within a frame are encoded concurrently across multiple threads. A CTU row can be encoded provided the CTU row immediately above it (top neighbouring row) is at least two blocks ahead of it in the encoding process [93, 7] Fig. 4.2 illustrates WPP whereby multiple CTU rows are being encoded in parallel across multiple threads. The green CTU's are the ones that are currently being processed. The transcoding speedup from WPP is more prominent for higher resolution videos because they have more CTU rows within a frame and thus more threads can take part in the encoding process.

- *Slices:*A frame can be divided into independent slices [92] and these slices can be processed concurrently across multiple cores/threads [94]. However, each slice has an associated header which increases the overheads and spatial prediction is poor at slice boundaries since the prediction is restricted within the slice block as inter-slice block referencing is not possible [95, 94].

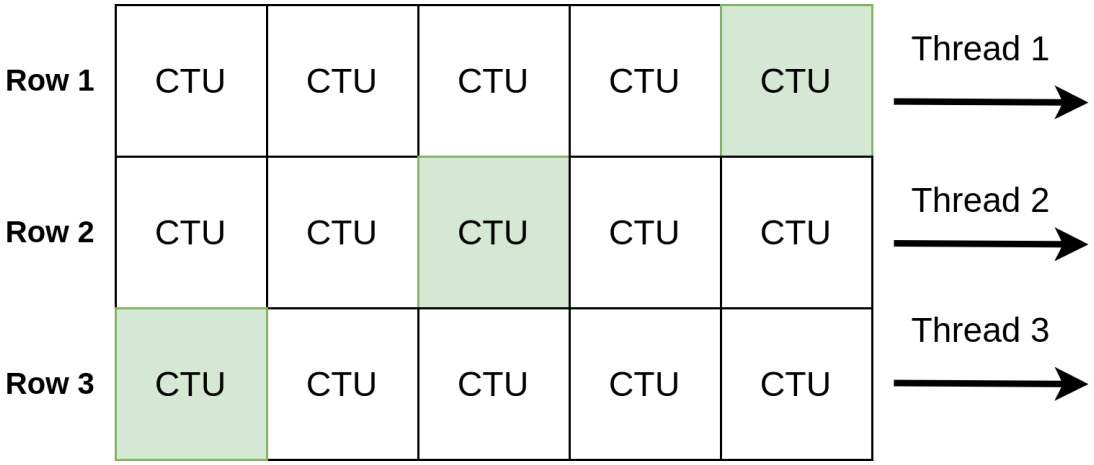


Figure 4.2: WPP in action: 3 threads encoding 3 CTU rows in parallel [7, 8, 9, 10, 11, 12]

Apart from the parallel encoding modes offered by HEVC, another important parameter which impacts the transcoding performance is preset. Preset governs the trade-off between transcoding speed and the transcoded video quality [96]. `libx265` encoder has the preset options of `ultrafast`, `superfast`, `veryfast`, `faster`, `fast`, `medium`, `slow`, `slower`, and `veryslow` (from fast to slow) [97]. The slower the preset, the better the quality per bit in the encoded video [97].

In addition to speeding up the transcoding process to meet real-time requirements, it is also important to generate good quality transcodes with minimum encoding bits. In general, the more encoding bits less is the information lost due to compression and thus more is the visual quality of a video. Rate control is a crucial step carried out by the transcoder and rate control determines the allocation of bits in a video sequence. The objective of rate control is either to encode videos at a certain target bitrate (bitrate control schemes) irrespective of the visual quality or to encode videos at a certain quality (quality control schemes) [98]. Bitrate

control schemes like Constant Bitrate (CBR) and variable Bitrate (ABR) always encode a video at a certain target bitrate. However, the optimal target bitrate can vary for different video sequences and this can lead to over-allocation of bits for hard to encode sequences and under-allocation of bits for easy to encode sequences. The concept of target bitrate forces the transcoder to stuff bits to meet a certain target bitrate requirement instead of balancing the trade-off between bitrate and visual quality for the video sequence. A quality control scheme like Constant Rate Factor (CRF) [99] on the other hand adjusts the encoding bits on a per frame basis to reach a target quality level and generates videos that have the same visual quality throughout. Additionally, CRF is also faster as it is a single pass technique. For the `libx265` HEVC encoder, the CRF values can range from 0 to 51 with lower CRF values providing better visual quality at the cost of larger file sizes.

4.2 Video Codec Analysis

`libx265` is a free HEVC encoder and for our research, we used `libx265` as the transcoding codec. Given the variety of parameters offered by `libx265` to speed up the transcoding process, this section investigates the impact of the most significant parameters on the transcoding speedups.

4.2.1 Methodology

We selected Big Buck Bunny [100], Crowd Run [101] and AWS Elemental Suzie [102, 103] videos at 2160p resolution as our bench-marking videos. These videos vary in content type, frame rates, and compression standards they are available in and hence represent the diverse forms in which video content is stored and streamed. The properties of these original videos are illustrated in Table 4.1. Using FFmpeg, we decoded the videos into raw uncompressed YUV 4:2:0 format (if they were originally encoded). We then segmented the raw videos into 2 second segments. Additionally, we downscaled the 2160p raw videos to several lower

Table 4.1: Properties of the Original Video Files

Video	Resolution	Format	Bitrate (kbps)	FPS
Big Buck Bunny	3840x2160	MP4	7498	30
Crowd Run	3840x2160	Y4M	4976642	50
AWS Suzie	3840x2160	MOV	4242490	24

Table 4.2: Transcoder Parameters Studied

Transcoder Parameter (libx265)	Value
Preset	Slow, Medium, Fast, Ultrafast
Inter Frame Parallelism	Enabled, Disabled
Wavefront Parallel Processing (WPP)	Enabled, Disabled
Slices	1,2

resolutions as well. Specifically, we downscaled the videos to 1080p, 720p, and 480p resolutions. These 2 seconds raw videos at resolutions of 2160p, 1080p, 72p, and 480p form our bench-marking inputs which would be transcoded to HEVC by the `libx265` encoder.

We transcoded the raw input segments to HEVC at multiple representations (quality levels) since DASH requires the video content to be made available in a variety of representations for different client needs. We transcoded the raw 2160p segments into 2160p, 1080p, 720p, and 480p HEVC variants. Similarly, we transcoded raw 1080p segments to 1080p, 720p, and 480p encoded variants, 720p raw segments to 720p and 480p encoded variants and finally raw 480p segments to 480p encoded variants. We used CRF as the rate control mode as it keeps the video quality constant throughout the segment. Specifically, we used a CRF value of 28 (default for HEVC) as it gives a good visual quality at an acceptable bitrate. For each of the conversions denoted by $s \rightarrow t$ where s is the source segment and t is the transcoded representation, we performed the transcodings under a variety of transcoder configurations as outlined in Table 4.2

4.2.2 Results

In this section we present the observations for the various transcoding tasks carried out. For brevity we present the results for Big Buck Bunny Video segments but similar results hold

true for the other two videos as well.

Impact of Preset: Fig. 4.3 shows the average transcoding times for the slow, medium, fast and ultrafast presets. For any source to target transcoding ($s \rightarrow t$), the preset order as per decreasing transcoding time is slow, medium, fast, and ultrafast preset. A slower preset spends more CPU cycles searching for similarities among pixels in a video segment leading to increased transcoding times. The impact of the preset is more important for high resolution transcodings like 2160p $\rightarrow$ 2160p where the slow preset takes about 13 seconds to transcode a segment whereas the ultrafast preset takes about 3.5 seconds to transcode a segment.

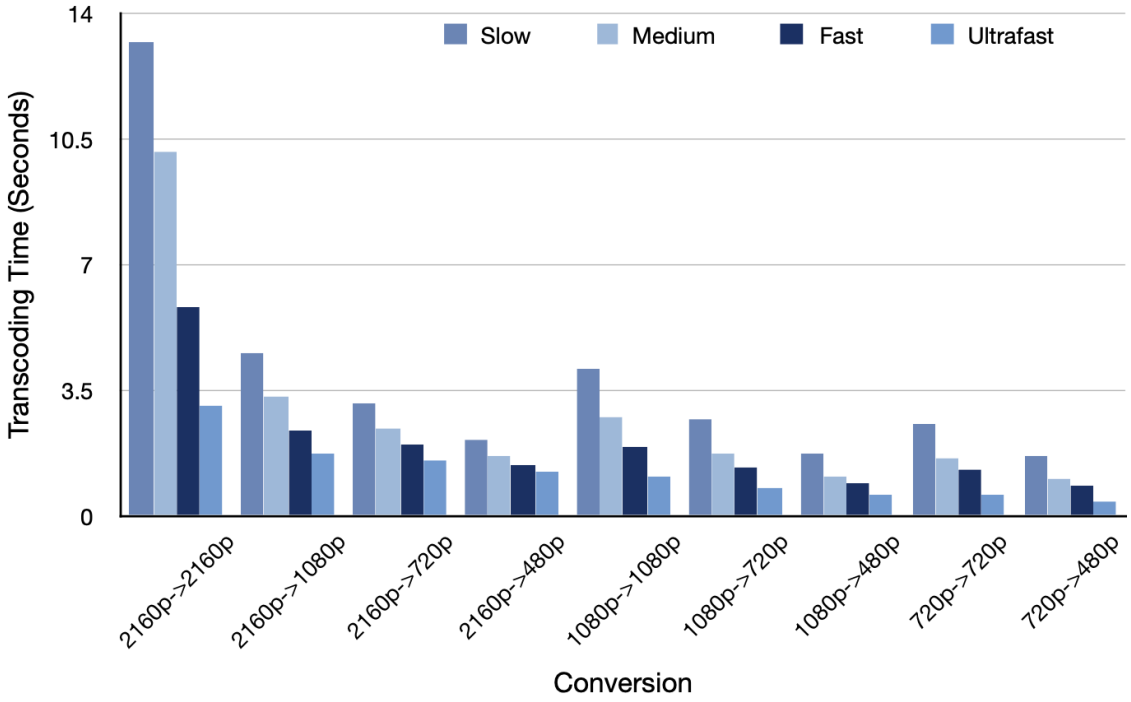


Figure 4.3: Relation between preset and transcoding time

Impact of WPP and inter-frame parallelism: Fig. 4.4 shows the average transcoding time for different combinations of inter-frame parallelism and WPP mode. The transcoding time is the highest when both WPP and inter-frame modes are disabled. Under such a configuration a single frame and a single CTU row is encoded at a time making the process extremely slow, especially for high resolution segments which have more CTU rows. The transcoding time goes down when either WPP or inter-frame parallelism is enabled. Enabling WPP gives the

lowest transcoding time, the impact being more visible for higher target resolutions due to the presence of a higher number of CTU rows at higher resolutions. The transcoding time when WPP and inter-frame mode are enabled together is similar to the case when only WPP is enabled. Since inter-frame parallelism and WPP share the CPU threads, they compete for CPU cores [38] and enabling both at the same time does not yield any significant transcoding speedups. Additionally, the dependency chain among CTU blocks gets more complex in the presence of both inter-frame parallelism and WPP which further restricts the transcoding speedups [92].

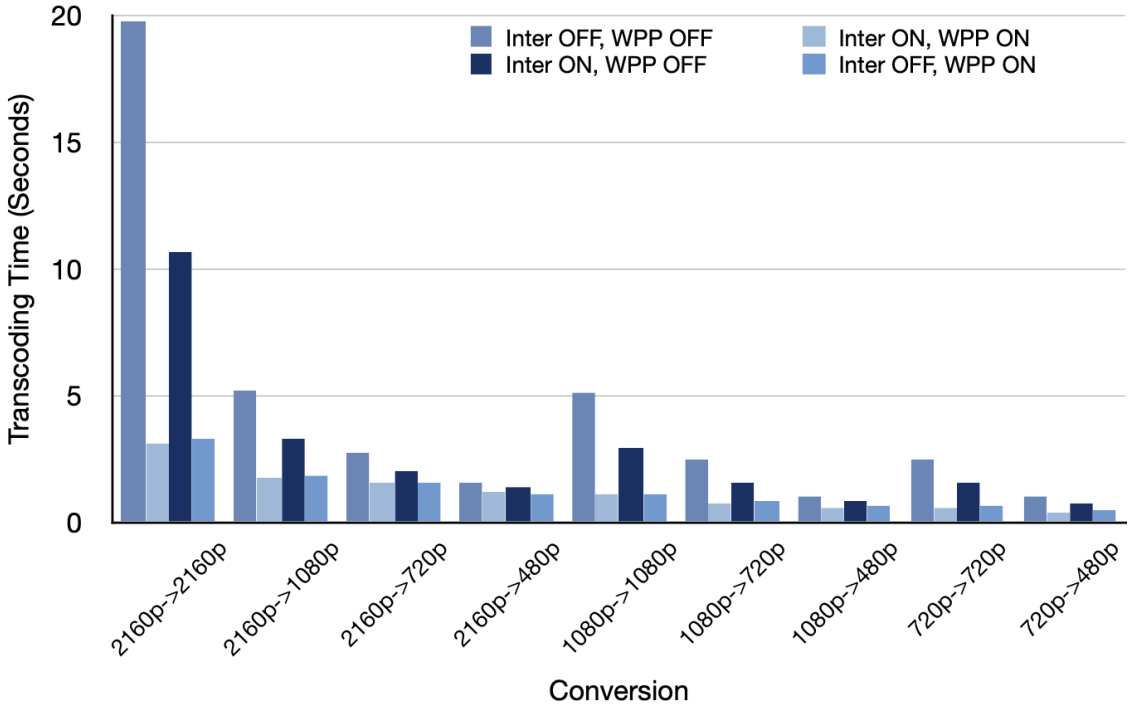


Figure 4.4: Relation between Libx265 parallelism modes and transcoding time

Impact of Slices: For the libx265 encoder, WPP has to be enabled to enable Slice based parallelism. Fig. 4.5 compares the transcoding time for a single slice and for two slices respectively. The transcoding time is similar for the two scenarios implying that multiple slices are not providing any transcoding speedup. For the `libx265` encoder slices can enhance the encoder performance only if other parallelism modes (such as inter-frame-parallelism and WPP) are not able to fully utilize the underlying resources. [92].

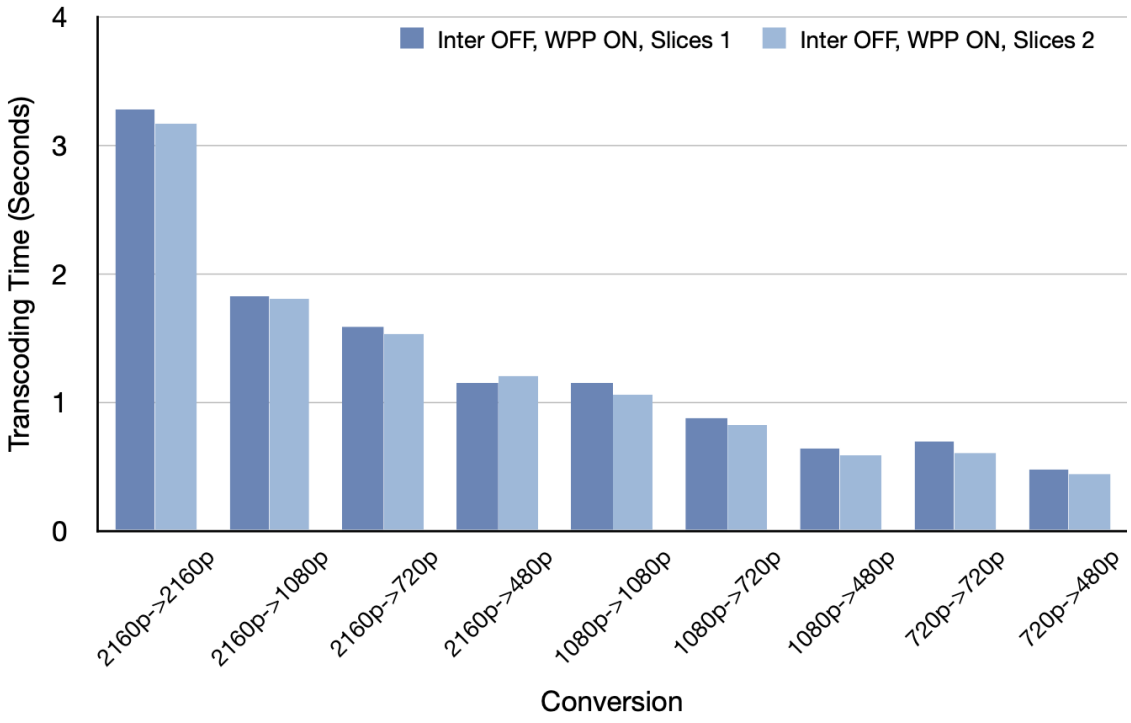


Figure 4.5: Relation between slices and transcoding time

4.2.3 Meeting the live rate

A real-time transcoder can sustain a live stream if the transcoder throughput (the number of frames transcoded per second) is at least as high as the video frame rate. For video segments d seconds long this implies that the transcoder must transcode the segment in a time less than or equal to d seconds. For high resolution and high quality transcoding it might not be possible to meet the live transcoding rate and in such scenarios multiple video segments must be transcoded in parallel to increase the effective transcoding rate. Therefore, it becomes important to evaluate the performance of the transcoder when multiple segments are transcoded in parallel on a system. For a WPP enabled and inter-frame parallelism enabled mode, the per-segment transcoding time increases as the number of concurrently transcoded segments increases as shown in Fig. 4.6. Whereas for a WPP disabled and inter-frame parallelism enabled mode, the per-segment transcoding time stays constant even when the number of concurrently transcoded segments increases to 6 as shown in Fig. 4.7. If we

compare the CPU utilization for the two scenarios, we observe that the per-segment CPU utilization is significantly higher when WPP is enabled as shown in Fig. 4.8 as opposed to when it is disabled as shown in Fig. 4.9. The high per segment CPU usage for WPP enabled mode increases the per segment transcoding time when multiple segments are transcoded in parallel.

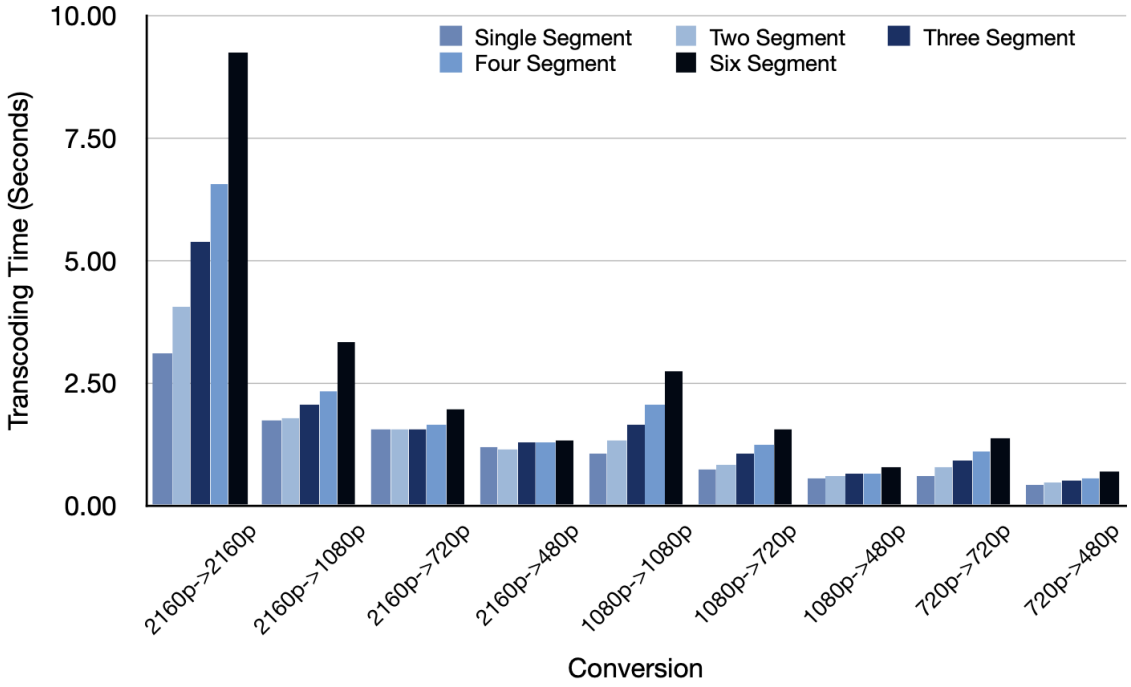


Figure 4.6: Per-transcoder transcoding time as a function of number of segments transcoded concurrently under WPP enabled and inter-frame parallelism enabled mode

4.3 Dockerized Transcoder

We propose a containerized transcoder to meet the following goals:

- *Cloud Deployment:* Video transcoding is often carried out on the cloud or edge servers and a containerized transcoder can be deployed with ease on the cloud thereby leveraging a large amount of computational resources in a scalable manner.

- *Distributed Transcoding:* The computational resources of a single system may saturate

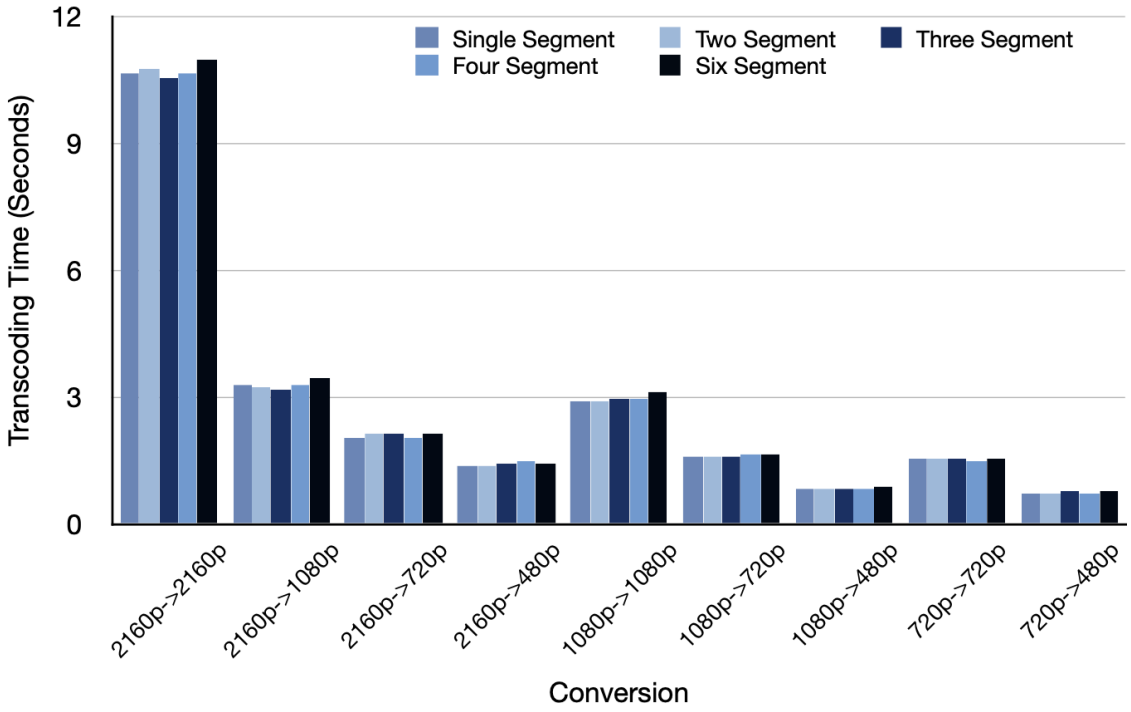


Figure 4.7: Per-transcoder transcoding time as a function of number of segments transcoded concurrently under WPP disabled and inter-frame parallelism enabled mode

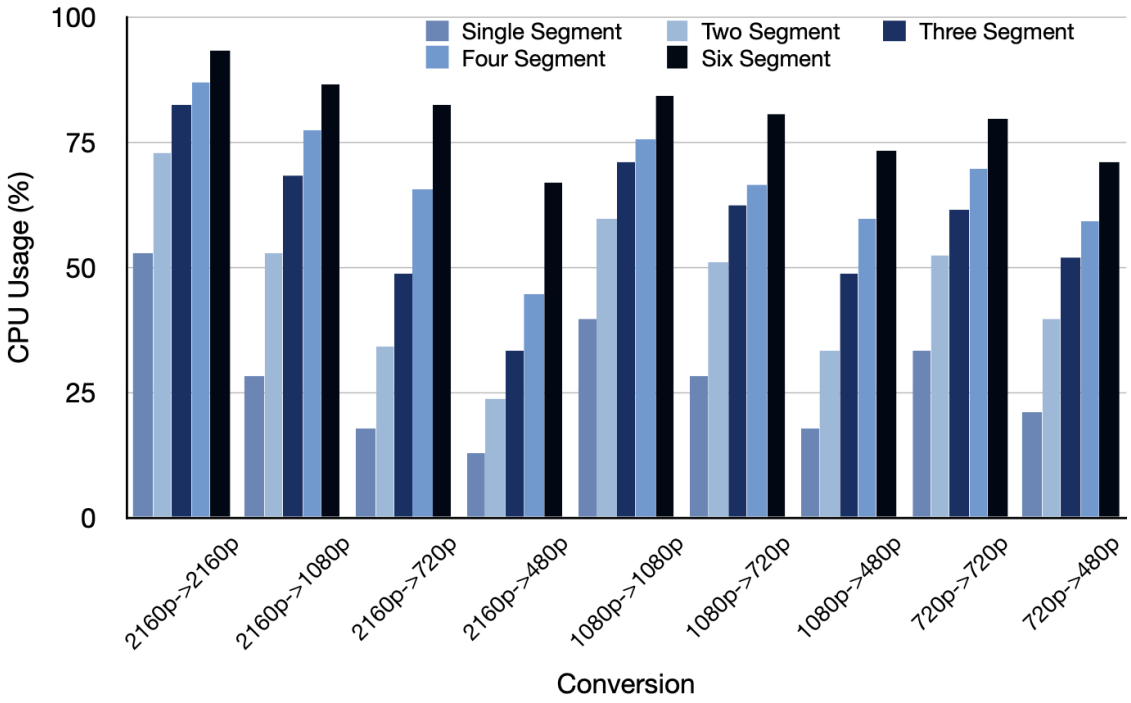


Figure 4.8: CPU utilization as a function of number of segments transcoded concurrently under WPP enabled and inter-frame parallelism enabled mode

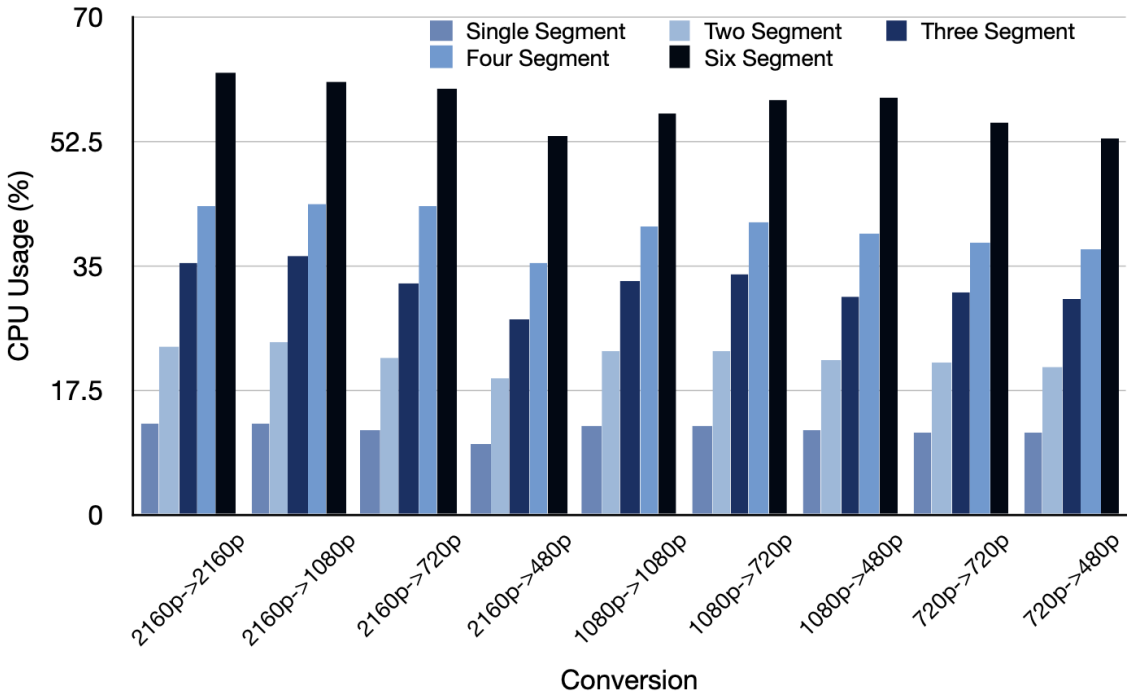


Figure 4.9: CPU utilization as a function of number of segments transcoded concurrently under WPP disabled and inter-frame parallelism enabled mode

when multiple transcoding jobs are executed concurrently as reported in Sec. 4.2.3. Hence there may be a need for distributed video transcoding across multiple systems to match the live video rate with minimum transcoding delay. A containerized transcoder can be deployed with ease on different distributed infrastructures.

- *Resource Efficiency:* Resources *e.g.,* cores allocated to a container can be fine tuned to achieve a certain transcoding throughput [41].

The first step towards creating a containerized transcoder is to build a docker image with all the required video processing tools installed. To build such an image we fetched the Ubuntu 18.04 image from the docker hub repository (A repository for hosting docker images) as the base image and installed `FFmpeg` compiled with `libx265` HEVC encoding library on top of the base operating system. Once built, the docker image can be shared across different infrastructures and containers can be created within milliseconds. Each container is a transcoder in itself and has a dedicated OS along with all the necessary video

transcoding dependencies.

Data within a container persist as long as the container exists. Hence a video segment transcoded by a container would be lost if the container is brought down. To overcome this limitation, we used bind mounts [104] to persist transcoded videos even after the container is removed. A bind mount is a shared storage space on the host system which can be connected to a container and the container has read-write access to the space. Hence a container can read the source video file (video which needs to be transcoded) directly from the shared storage and write the transcoded video file directly to the shared space as shown in Fig. 4.10. Not only does this persist the transcoded videos but also prevents additional transmission overheads of transmitting videos to and from a container.

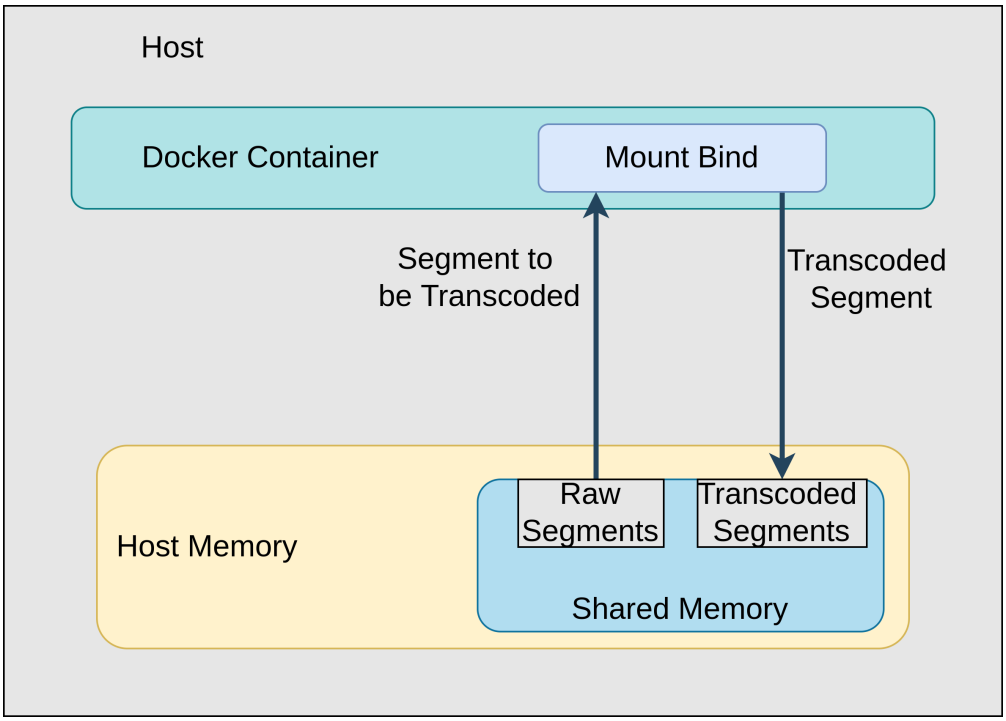


Figure 4.10: Docker Bind Mount Architecture to Get Read-Write Access to the Host Filesystem

To transcode a video using containers, a container is launched from the docker image. Transcoding instructions along with the transcoding parameters are passed as arguments to the container. Once a container finishes executing the transcoding job and writes the

transcoded video file to the shared memory, the container is deleted. Thus a new container is created for each video transcoding job. With docker, it is possible to control how many CPU cores a container can use [41]. The extent of transcoder parallelism (WPP and inter-frame parallelism) and consequently the per-transcoder throughput can then be precisely controlled by tuning the cores allocated to the container. Fig. 4.11 shows the relationship between CPU cores allocated to the container and transcoding time for the bugs bunny segments when transcoded with a WPP enabled and inter-frame parallelism enabled mode. From the figure, we can conclude that the transcoding speedup saturates with increasing cores and the saturation core is a function of source and target resolutions. This relationship was even published by Sameti *et al.* [38].

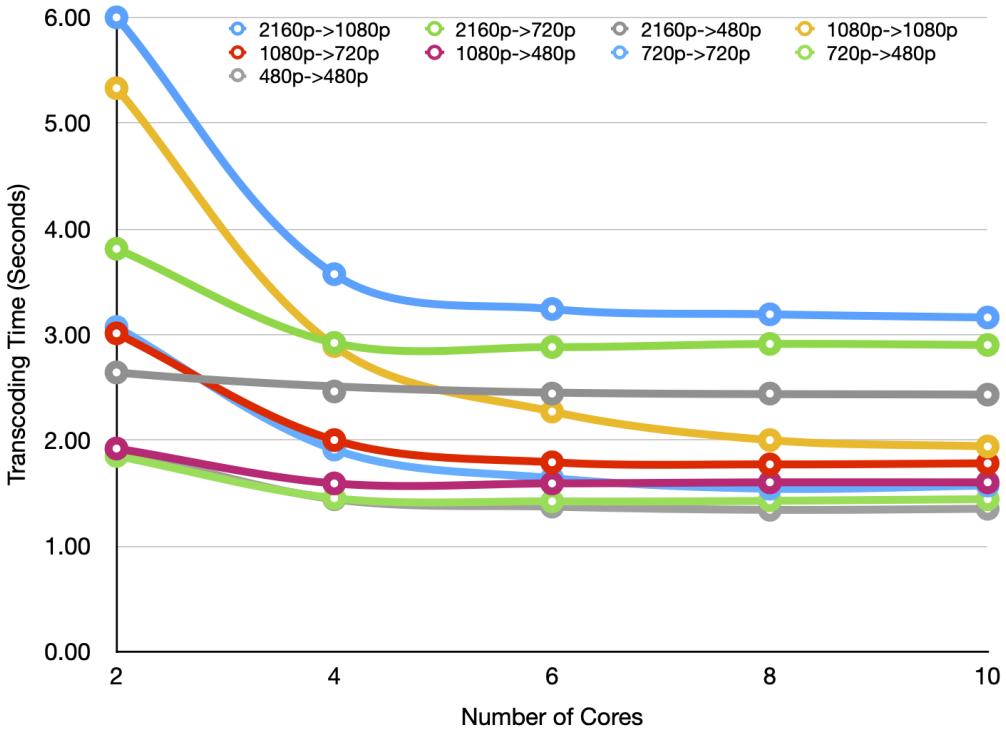


Figure 4.11: Relation between CPU cores allocated to the container and the transcoding time for WPP enabled configuration

4.4 Summary

In this chapter, we summarized the High Efficiency Video Coding Standard. We studied WPP, inter-frame parallelism, slices, and presets as tools offered by the codec to speed up the transcoding process. Our results show that WPP and inter-frame parallelism significantly speed up the transcoding process by encoding multiple frames/CTUS concurrently across multiple CPU threads. In particular, the transcoding speedups achieved with WPP is the highest and thus for real-time transcoding WPP must be enabled. We also evaluated the impact of presets on the transcoding speed and concluded that slower presets are simply too slow to be employed for real-time transcodings. We also evaluated the impact of transcoding multiple segments concurrently on a machine. With WPP, per transcoder performance deteriorates as the number of concurrently transcoded segment increases as the segments compete for CPU resources. However with inter-frame parallelism, the deterioration per transcoder is much less with an increase in the number of concurrently transcoded segments.

We then provided an overview of containerized transcoder. Containers are extremely lightweight and portable which makes them easy to deploy on any given hardware with minimum modifications. Additionally, resources (cores) allocated to a container can be fine-tuned to precisely control the transcoder throughput and make the transcoding process resource efficient. We detailed the steps for creating a containerized transcoder.

The analysis carried out in this chapter help in directing the research carried out in the subsequent chapters. Specifically, from the analysis carried out in this chapter, we can conclude that WPP must be enabled to achieve real-time performance and with dockers, it is possible to fine-tune the resources allocated to a container which can facilitate a more efficient resource utilization.

Chapter 5

Balancing Performance and Resource Efficiency in Real-time Transcoding

Transcoding is a computationally intense process especially for videos with high resolution (4K and beyond) and high frame rates and can incur significant operational costs making it expensive for content providers. Thus, to achieve low delays and high Quality of Experience (QOE), it is essential to speedup the transcoding process while keeping it resource efficient. In this chapter we aim to balance this conflicting goal of faster transcoding and resource savings.

Vertical scaling (increasing the number of resources *e.g.,* cores) seems like an effective strategy to speed up the transcoding process as transcoding is a highly resource demanding process. However, because of the spatial and temporal dependencies which exist among blocks of a video frame(s) there is an encoding parallelism limit and transcoding time saturates with increasing CPU cores [38] and a further increase in CPU cores is unlikely to provide any transcoding speedups. Hence vertical scaling is sub-optimal and if not performed efficiently will lead to resource over-provisioning and increased operational costs for cloud based transcoding services.

Containerized transcoding solutions provide a mechanism for balancing the conflicting

goals of faster transcoding and resource efficiency in the context of live streaming. Containers can be deployed on the cloud thereby leveraging a large amount of computational resources in a scalable manner. Furthermore, resources, *e.g.,* cores, allocated to a container can be controlled in a fine-grained manner to achieve efficiency [41, 38]. Though this scheme yields good speedup while offering the ability to fine tune the resource allocation, there is one essential challenge - determining the minimum size of the container (*i.e.,* the minimum number of cores) to achieve a certain transcoding rate. Even more challenging, the container size will vary not only for different videos, but also from segment to segment. For instance, a complex segment may need more cores to achieve a certain transcoding rate than a simple segment.

To address this challenge, we need to balance the performance and resource efficiency segment by segment. In other words, we will seek the right resource provisioning (neither under-allocation nor over-allocation) to achieve a target transcoding rate for each segment. From literature review outlined in Sec. 3.1.1 and our own empirical analysis, we learn that transcoding rate and resource requirement are highly correlated with segment properties such as resolution, frame rate, duration, segment complexity and the transcoding parameters. Little work has been proposed on leveraging these insights into a concrete mechanism for tuning the transcoding of individual segments in the context of live streaming.

Our solution offers machine learning (ML) based models to guide the per segment transcoder tuning. We explore the features that these models need to yield accurate per-segment predictions. We also outline how they can be obtained in a lightweight manner during the transcoding process thus minimizing the computational overheads and making it possible to make the predictions in real time. Based on these features, we developed two distinct ML models. Given a fixed number of cores, the first regression model predicts the achievable transcoding time based on segment properties and transcoding parameters. The second classification model predicts the minimum number of cores needed to achieve a target transcoding rate. Combining the two models, we propose an algorithm for balancing the performance

(*e.g.,* the transcoding time) and resource requirement (*e.g.,* the number of cores) for any given segment.

This chapter is organized as follows. In Sec. 5.1 we first present the input dataset which can be used to build the machine learning models. We then present a general analysis of the dataset by means of exploratory data analysis in Sec. 5.2. In Sec. 5.3 we show how we used AutoML to generate accurate prediction models. In Sec. 5.4 we evaluate the performance of the prediction models. Finally, in Sec. 5.6 we show how the transcoding time prediction model, and the CPU cores prediction model can be used in conjunction to balance performance and resource efficiency for real time transcoding.

5.1 Dataset Generation

A good dataset is essential for training a good prediction model. In this section we first outline the video selection step followed by the feature extraction and segment transcoding steps.

5.1.1 Video Selection

To craft a qualitatively and quantitatively rich transcoding dataset, an important requirement is to select several videos having diverse complexities. Videos vary in the kind of content they hold, and the transcoding performance depends on the video contents. We rely on Spatial Information (SI) and Temporal Information (TI) as complexity indicators with the former measuring the similarity among neighbouring pixels within a frame and the latter measuring the motion across video frames. We selected a total of 15 Ultra HD (4K) videos from the UVG Dataset [105], the AWS Elemental Dataset [102, 103] and the Xiph Derf Dataset [101]. All the videos were uncompressed to YUV:420 format and encapsulated in a Y4M container. Fig. 5.1 shows the SI-TI(complexity) distribution for the selected videos and Table 5.1 summarizes the properties of the video sequences.

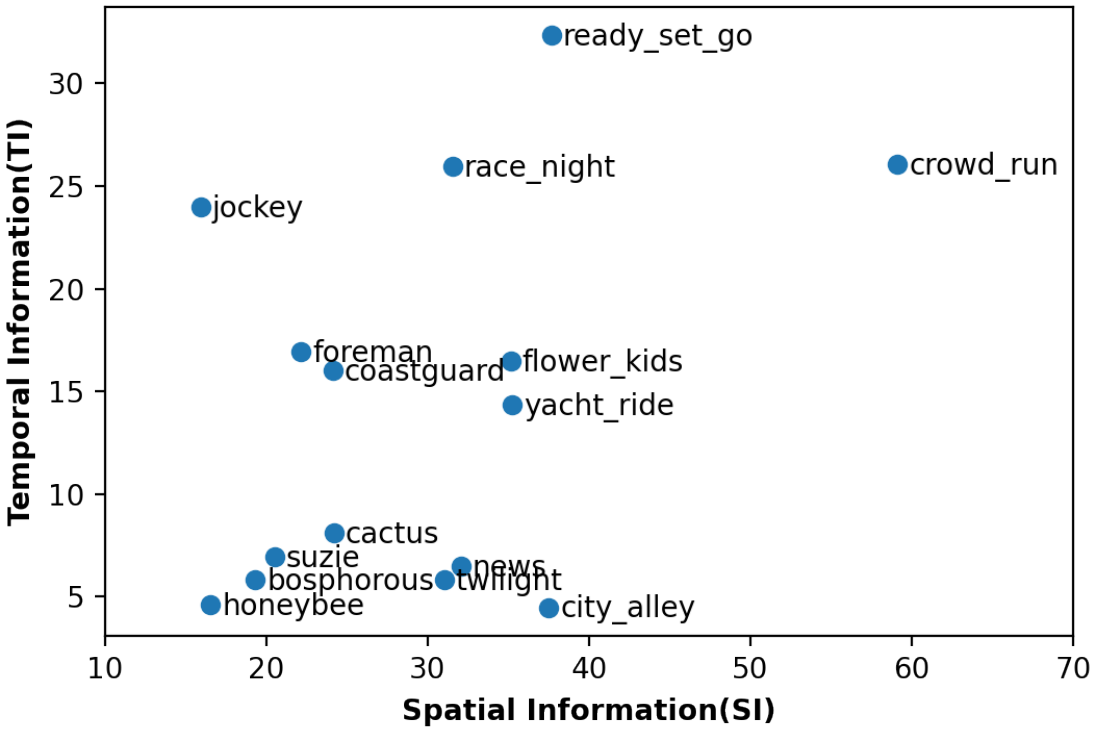


Figure 5.1: SI-TI Distribution for the original 4K videos

Table 5.1: Original video characteristics used for generating the prediction data

Video	**Creator**	**Resolution**	**FPS**	**Bitrate (kbps)**	**Codec**	**Duration (Seconds)**	**# 1 Second Segments**	**# 2 Second Segments**
Cactus	AWS Elemental	3840x2160	24	2388788	YUV 4:2:0	10	10	5
Coastguard	AWS Elemental	3840x2160	24	2388788	YUV 4:2:0	10	10	5
Suzie	AWS Elemental	3840x2160	24	2388788	YUV 4:2:0	14	14	7
Foreman	AWS Elemental	3840x2160	24	2388788	YUV 4:2:0	10	10	4
News	AWS Elemental	3840x2160	24	2388788	YUV 4:2:0	8	8	4
Jockey	UVG Dataset	3840x2160	30	2985985	YUV 4:2:0	20	20	10
Ready Set Go	UVG Dataset	3840x2160	30	2985985	YUV 4:2:0	20	20	10
Honey Bee	UVG Dataset	3840x2160	30	2985985	YUV 4:2:0	20	20	10
Bosphorous	UVG Dataset	3840x2160	30	2985985	YUV 4:2:0	20	20	10
Yacht Ride	UVG Dataset	3840x2160	30	2985985	YUV 4:2:0	20	20	10
Race Night	UVG Dataset	3840x2160	50	4976642	YUV 4:2:0	12	12	6
Twilight	UVG Dataset	3840x2160	50	4976642	YUV 4:2:0	12	12	6
City Alley	UVG Dataset	3840x2160	50	4976642	YUV 4:2:0	12	12	6
Flower Kids	UVG Dataset	3840x2160	50	4976642	YUV 4:2:0	12	12	6
Crowd Run	Xiph Derf	3840x2160	50	4976642	YUV 4:2:0	10	10	5

Since a raw uncompressed video has a large file size, videos are encoded at the source prior to transmission to the transcoding server. We therefore encoded the videos to AVC using `FFmpeg` and `libx264` library at resolutions of 4K and 2K. We used AVC as the encoding codec as it is one of the most popular and widely supported codec for online streaming. We then split the encoded videos into 1 second and 2 second segments. Segments whose duration is 1 or 2 seconds are common for live streaming applications [106]. We had a total of 628 input segments (210 one second segments at 4K + 210 one second segments at 2K + 104 two second segments at 4K + 104 two second segments at 2K). These compressed (AVC encoded) 1 and 2 second segments form our inputs, which will be transcoded to HEVC at multiple representations (quality levels) to generate the transcoding dataset.

5.1.2 Feature Extraction

The next stage in the process of crafting the dataset is to extract relevant features which impact the transcoding performance *i.e.,* transcoding time. According to the literature review presented in Sec. 3.1.1 and our own observations, the transcoding time and resource requirement are highly related to segment properties, the codec used and the underlying hardware. Therefore, we broadly classify the prediction features into three categories:

- *Segment Level Features*: It includes the segment complexity (SI and TI values), segment duration, frames rate (fps) and the video resolution.
- *Transcoding Level Features*: It includes the codec employed, codec parameters along with other transcoding parameters.
- *Infrastructure Level Features*: It includes the clock speed of the CPU and the number of CPU cores allocated to the transcoder.

To make real time predictions possible it is important to extract the prediction features in real time with minimum overheads. Apart from segment complexity (SI and TI), the other features on which the transcoding time depends are either readily available (or can be

readily extracted in real time like frame rate, segment duration and video resolution) or are parameters specified by the user (codec, transcoding parameters, target resolution, number of CPU cores). Computing the SI and TI values requires scanning the video pixel by pixel and the process can be time consuming especially for high resolution 4K segments.

To make the SI and TI extraction in real time possible, we transcoded each input segment to a 240p low resolution downscaled variant using `libx264` at a CRF value of 22 (any other codec can be used as well) and extracted the SI and TI values [23] along with the resultant bitrate [77] for the 240p downscaled variant as segment complexity features (as suggested by [23, 77]). Transcoding to a low resolution like 240p takes little time and extracting SI and TI from the 240p variant is faster as far less pixels need to be scanned compared to the original high resolution segments [23]. There exists a strong correlation between the SI and TI values of the original high resolution segments and its downscaled version [23]. SI is the measure of information density in a frame and when a video is downscaled the information contained in it is still the same but confined to a much smaller area. As a result, information density increases, leading to an increase in SI score [76]. TI on the other hand is a measure of motion and is therefore independent of the video resolution [76]. We used `siti` tool [107] to compute the spatial and temporal information of the 240p downscaled segments When we downscaled the video to 240p resolution, we even extracted the resultant bitrate from the 240p variant as a complexity indicator as well [77]. CRF is a variable bitrate encoding where the bits spent by the encoder depends on the video contents (complexity) and therefore the 240p bitrate is a measure of segment complexity as well. Throughout the remainder of this thesis, 240p SI, 240p TI and 240p bitrate value will be referred to as downscaled SI, downscaled TI and downscaled bitrate, respectively.

For each of the input segments we recorded the segment name, frame rate (fps), segment duration, segment resolution, downscaled SI, downscaled TI, downscaled bitrate as segment level features. Furthermore, we recorded the segment height (in pixels) as an additional segment level feature as the number of CTU rows depend on the height of the frame.

Table 5.2: Table Describing the features selected for the predictions

Feature	**Value**	**Significance**
Duration	Numeric	The number of frames to be transcoded is proportional to the segment Duration
FPS	Numeric	The number of frames to be transcoded is proportional to the FPS value
Input Height	Numeric	The Number of CTU rows in the source segment is proportional to the Input Height
Output Height	Numeric	The Number of CTU rows in the transcoded segment is proportional to Output Height
Input Pixels	Numeric	The total number of pixels to be transcoded in the source segment
Output Pixels	Numeric	The total number of pixels to be encoded in the target segment
Downscaled SI	Numeric	Measures pixel similarity in a frame
Downscaled TI	Numeric	Measures movement across the frames
Downscaled Bitrate	Numeric	Bits needed to encode the downscaled variant
Preset	Category	Determines the transcoding speed
Number of Cores	Numeric	The processing resources available to the transcoder

From literature review and from our own analysis, we observe that the transcoding time of a video segment depends on duration (as suggested by [23, 77, 74, 75]), fps (as suggested by [23, 77, 70]), input height, output height (as suggested by [23, 77]), input pixels, output pixels (as suggested by [23, 77]), downscaled SI (as suggested by [23]), downscaled TI (as suggested by [23]), downscaled bitrate (as suggested by [77]), preset (as suggested by [23, 77]) and number of cores (as suggested by [38]). Table 5.2 shows the features we selected to train our prediction models.

Table 5.3: Transcoding Parameters

Parameters	Value(s)	Description/Significance
Codec	High Efficiency Video Coding (HEVC)	The video codec for compressing the video
Paralleism Mode	Wavefront Parallel Processing (WPP)	WPP allows CTU rows in a frame to be processed in parallel thus speeding up the transcoding process
CRF	28	CRF determines the quality level of the transcoded segments. It adjusts the encoding bits based on the segment complexity to reach a target quality level
CPU Cores	2,3,4,5,6,7,8, 10,12,14	The number of CPU available to the transcoder enabling parallel encoding of CTU blocks.
Preset	Ultrafast, Veryfast	Preset determines the trade off between transcoding speed and video quality (or file size) [96].
Source ->Target Resolution	2160p ->2160p, 2160p ->1080p, 2160p ->720p, 2160p ->480p, 1080p ->1080p, 1080p ->720p, 1080p ->480p,	Source Resolution and Target resolution determines the number of pixels to be processed. Additionally number of CTU rows also depend on the frame dimensions

5.1.3 Segment Transcoding Stage

For the transcoding task we resorted to HEVC as the codec. We employed CRF as the rate control mode since it is a single pass technique and generates constant quality encodes. We used a CRF value of 28 (default for HEVC) for the transcoder. Furthermore, we enabled WPP to maximize the transcoder throughput and used `ultrafast` and `veryfast` presets and avoided the slower presets. All the selected transcoding parameters comply with low delay transcoding. The transcoding parameters to generate the dataset are summarized in Table 5.3

For running the transcoding tasks, we created a Docker image configured with `FFmpeg` and `libx265` HEVC encoder library. We then launched containers from it on a virtual machine (VM) in the Cybera Cloud [108] and ran transcoding scripts from within the container to transcode input video segments under a set of transcoding parameters. The VM instance was configured with 16 cores, 32 GB of memory and running Ubuntu 18.04. The set of tunable parameters includes cores allocated to the docker container, target resolution and the preset. We transcoded the 2160p segments to 2160p, 1080p, 720p and 480p variants and the 1080p segments to 1080p, 720p and 480p variants, respectively.

For each of the transcoding task, we recorded the transcoding time and combined this information with the input segment properties, transcoding parameters and the CPU cores allocated to the transcoder to generate records for the dataset which can then be used to train the prediction model. The following fields are recorded for each of the transcoding record in our dataset: segment name, fps, segment duration, downscaled bitrate, downscaled SI, downscaled TI, source resolution (input width $\times$ input height in pixels), source height(pixels), target resolution (output width $\times$ output height), target height(pixels), preset, CPU cores and transcoding time. We had a total of 43960 records (314 4K segments $\times$ 4 target resolutions $\times$ 2 presets $\times$ 10 CPU core values + 314 2K segments $\times$ 3 target resolutions $\times$ 2 presets $\times$ 10 CPU core values).

5.2 Exploratory Data Analysis

In this section we report the general observations for the prepared transcoding dataset. The transcoding time for the dataset ranges from 1.46 seconds to 104.31 seconds with a mean transcoding time value of 4.90 seconds.

5.2.1 Impact of segment duration and fps

The transcoding time is proportional to the number of frames which need to be processed. The longer the segment, the more the frames contained in it resulting in higher transcoding times. The average transcoding time for 1 second and 2 second records was 4.08 seconds and 6.56 seconds, respectively. Similarly, the higher the fps the more the frames contained within each second of the video resulting in higher transcoding times. The average transcoding time for 24 fps, 30 fps and 50fps segments was 3.98 seconds, 4.61 seconds and 6.22 seconds, respectively.

5.2.2 Impact of preset

The transcoding time is highly dependent on the preset used. The average transcoding time for the ultrafast preset and veryfast preset was 3.90 seconds and 5.91 seconds, respectively. The throughput(*i.e.,* frames transcoded per second) for ultrafast and veryfast preset is comparable to the frame rate (fps) of the video segments while other presets are too slow to be employed for low latency applications like live streams.

5.2.3 Impact of segment complexity

Fig. 5.2 shows the relation between downscaled bitrate and transcoding time for 2 second long segments having fps value of 30 when transcoded from 1080p $\rightarrow$ 1080p with ultrafast preset on 8 cores. We observe that the transcoding time is proportional to the downscaled bitrate. Similar observation holds for other conversions as well but for brevity we only present the relationship for 1080p $\rightarrow$ 1080p conversion. The reason for this proportional behavior is attributed to CRF encoding. CRF generates constant quality encodes and the bits spent to encode a segment is proportional to the segment complexity. The more complex the segment, the more the CPU cycles are needed to transcode the segment resulting in higher transcoding times.

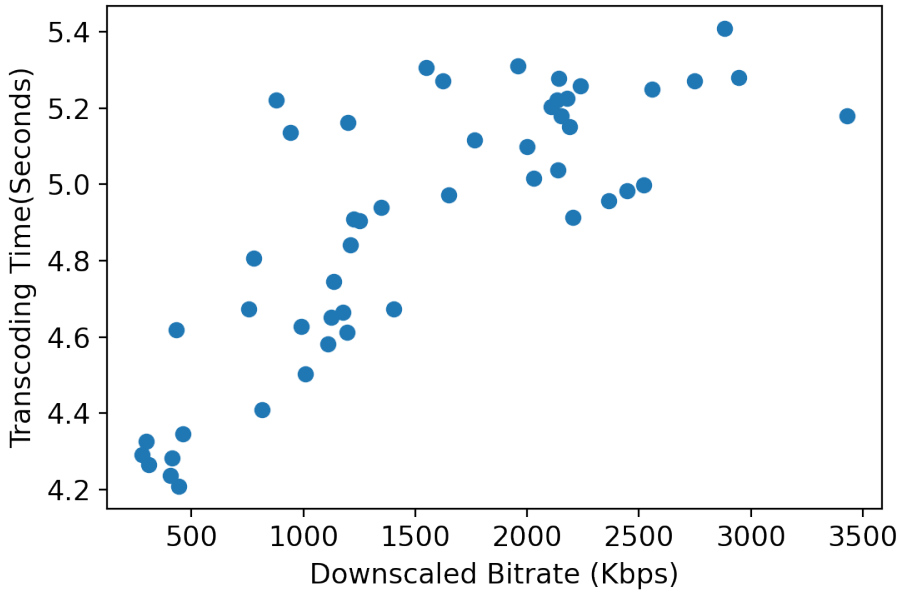


Figure 5.2: Relationship between downscaled bitrate and transcoding time for 2 second long segments having fps value of 30 when transcoded from 1080p → 1080p with ultrafast preset on 8 cores

Similarly, the transcoding time is also related to the downscaled SI and downscaled TI values. In general, the higher the downscaled SI and downscaled TI values higher is the transcoding time but there are exceptions and the relation is not a strictly proportional one as evident from Fig. 5.3.

5.2.4 Impact of Resolution

Transcoding involves decoding the source segment and encoding it again to a new desired variant. The decoding time depends on the source resolution and the encoding time depends on both the source resolution and the target resolution. Hence the higher the resolutions involved, the higher will be the transcoding time. For 2160p → 2160p transcoding the average transcoding time has was 12.11 seconds whereas for 1080p → 480p transcoding the average transcoding time was just 2.30 seconds. Table 5.4 depicts the average transcoding time for different conversions.

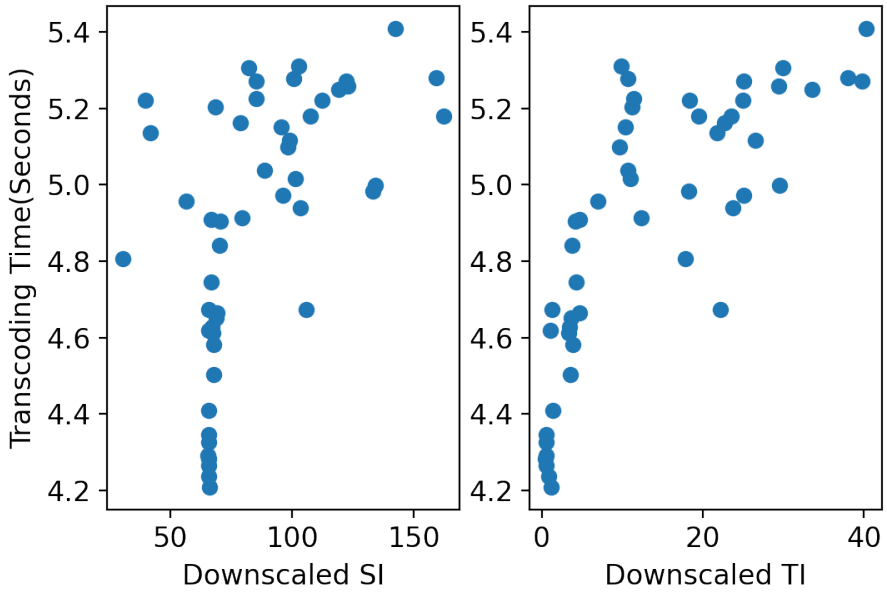


Figure 5.3: Relationship between downscaled SI-TI and transcoding time for 2 second long segments having fps value of 30 when transcoded from 1080p → 1080p with ultrafast preset on 8 cores

Table 5.4: Average transcoding time for different source → target transcodings

Conversion	**Transcoding Time (Seconds)**
2160p → 2160p	12.11
2160p → 1080p	5.33
2160p → 720p	4.13
2160p → 480p	3.18
1080p → 1080p	4.15
1080p → 720p	3.12
1080p → 480p	2.30

5.2.5 Impact of CPU Cores

The CPU cores determine the number of threads which can take part in the parallel transcoding of CTU blocks and impact the transcoding time. Fig. 5.4 shows the average transcoding time and Fig. 5.5 shows the average CPU utilization as a function of number of CPU cores for different source → target transcoding. From the figures it is evident that the transcoding time saturates with increasing CPU cores and a further increase in CPU cores doesn't yield any further speedup as was suggested by Sameti *et al.* [38]. This is because spatial and temporal dependencies exist among video blocks which restrict parallel encoding despite having the available resources. A CTU block cannot be encoded until the block on which it depends has been encoded and this chain of dependencies puts a limit on the maximum achievable parallelism [38, 92]. Vertical scaling (*i.e.,* increasing the number of cores allocated to the transcoder) is sub-optimal after a certain threshold number of cores. Additionally, the number of cores at which the transcoding speedup saturates value varies from segment to segment and is highly dependent on the source resolution and the target resolution. The higher the resolution, the more the CTU rows in a video frame which allows for more CPU threads (or more cores) to take part in WPP resulting in an increased CPU usage.

5.3 Prediction Models

In this section, we present our prediction algorithms. We develop two prediction models:

1. *Transcoding Time Prediction Model:* To predict the transcoding time of a segment given the segment properties, the transcoding parameters and the number of cores allocated to the transcoder.

2. *CPU Cores Prediction Model:* To predict the minimum number of CPU cores to be allocated to a transcoder to reach a target transcoding throughput.

To begin with we used `python` and `keras` [109] and developed manual ANN models

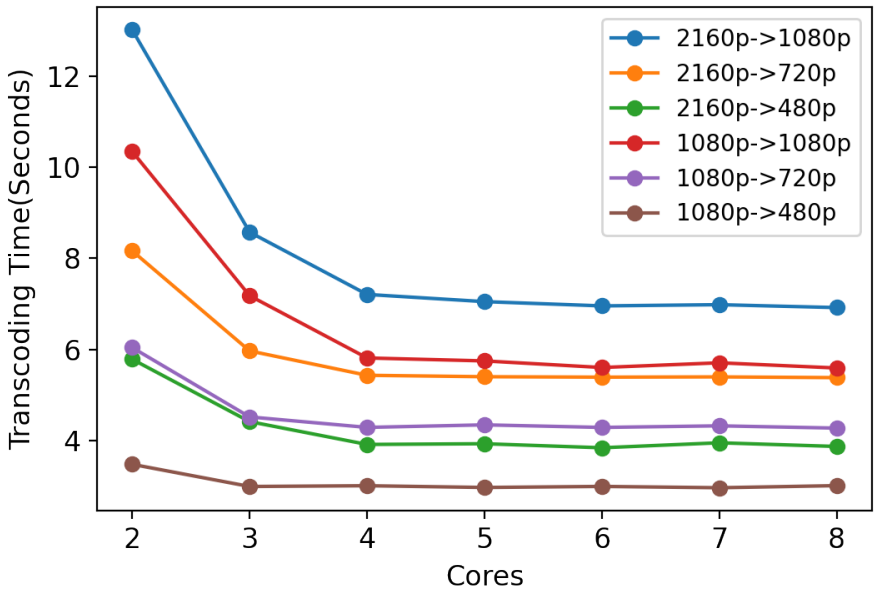


Figure 5.4: Relationship between number of CPU cores and transcoding time for 2 second long segments having fps value of 30 when transcoded with ultrafast preset

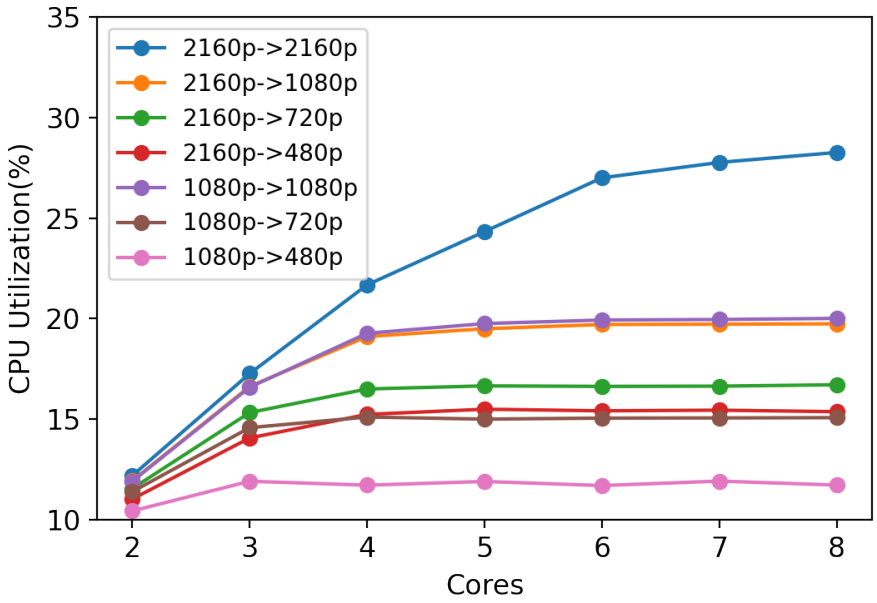


Figure 5.5: Relationship between number of CPU cores and CPU utilization for 2 second long segments having fps value of 30 when transcoded with ultrafast preset

for the two prediction tasks. A summary of the Keras models is presented in Appendix (Sec. A.2) We then automated the machine learning workflow using Google AutoML [110] and came up with prediction models that outperformed the manually developed models. For our research we resort to the AutoML models as they have a better performance than the manually developed prediction models.

In this section we first provide an overview of the AutoML tool used to generate the models. We then describe the design of the two prediction models along with the evaluation metrics used to examine the performance of the models.

5.3.1 AutoML

Fig. 5.6 depicts the standard stages of a machine learning workflow. The stages are briefly summarized:

1. Data Collection: Gather the relevant data on which the model will be trained. For our research the transcoding records form the data.

2. Data Preprocessing: Performed to get the most information out of the data. It involves steps like data cleaning and data transformation(feature engineering).

3. Model Generation: Decide the model to be used for mapping input-output relationship along with the model parameters.

4. Model Evaluation: Evaluate the performance of the model by examining the models performance metrics. Model generation and model evaluation is an iterative process.

5. Model Deployment: Perform real world prediction on unseen records.

Figure 5.6: Stages of a machine learning workflow

.

Building a machine learning (ML) pipeline requires extensive effort in preprocessing the data to ensure that the data is usable and then coming up with a model which best describes the relation between the features (independent variables like frame rate, source resolution, target resolutions, segment duration, downscaled SI and TI, and downscaled bitrate) and the target variable (*i.e.,* transcoding time and number of cores). Nonetheless, the learning process may still fail to capture the input-output relationship accurately either because the prepared data was of poor quality, or the model (or the model parameters) chosen was not the right one. With so many optimization choices right from the data preprocessing stage to the training stage, it is quite a challenge to come up with an accurate prediction model. Google AutoML [110] is a collection of ML products that automates the development of machine learning models. AutoML takes care of data pre-processing, model generation and model evaluation [111]. It trains the provided data on several different models and returns the best fit model or an ensemble of models along with a feature importance (correlation) matrix illustrating how much each feature contributes in making the final prediction.

Once the dataset in a suitable format (csv file) is given to AutoML, the data is divided into training, validation, and test sets [112]. The majority of the data falls in the training dataset and this data is used to generate (or train) the model. Validation set is used to find the best model hyperparameters during the training phase [113]. Finally, after the model hyperparameters are determined, the final model is tested against the unseen test set. Training, validation, and testing are all automated in google AutoML. By default, 80% of the transcoding data is used for training, 10% for validation and 10% for testing [112]. We provided our transcoding dataset to AutoML and let AutoML evaluate different ML models to find the best performing models for predicting the transcoding time and the number of cores.

5.3.2 Transcoding Time Prediction Model

We used 11 features to train the transcoding time prediction model - duration (as suggested by [23, 77, 74, 75]), fps (as suggested by [23, 77, 70]), input height, output height (as suggested by [23, 77]), input pixels, output pixels (as suggested by [23, 77]), downscaled SI (as suggested by [23]), downscaled TI (as suggested by [23]), downscaled bitrate (as suggested by [77]), preset (as suggested by [23, 77]) and number of cores (as suggested by [38]). Duration and fps determine the number of frames to be processed by the transcoder. Input and output pixels determine the total number of pixels to be processed. Input and output height determine the number of CTU rows and hence the extent of WPP. Downscaled SI, downscaled TI and downscaled bitrate determine the complexity of the video segment. Preset determines the tradeoff between quality and speed and number of CPU cores determines the processing resources allocated to the transcoder. Although some of the features such as Input Height and Input Pixels may look redundant/related (as input pixels is fixed for a given input height), we still decided to include the related features as AutoMl is smart enough to assign a lower prediction weight to related features (if indeed that's the case). From our dataset analysis (Sec. 5.2.5) we observed that for any transcoding record regardless of the segment properties or the target resolution, the transcoding time saturates at most with 8 cores as evident from Fig. 5.4. Hence, we decided to use a subset of the transcoding dataset for which the core value ranges from 2 to 8 (inclusive) and discard the higher core records for training our model. We had a total of 30772 records for training the dataset.

To train a model, Google AutoML needs an optimization metric based on the objective of the intended model. As our objective is to minimize the difference between actual transcoding time and predicted transcoding time, we used Mean Absolute Error (MAE) as shown in Equation 5.1 as the optimization metric. For evaluating the performance of the model, we used Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE) and r_squared (r^2) values. MAE as shown in Equation 5.1 represents the mean absolute error (in seconds) in predicting the transcoding time. RMSE as shown in Equation 5.2 is the root of the mean squared prediction error (in seconds). Mean Absolute

Percentage error as shown in Equation 5.3 represents the mean of the absolute percentage difference between actual transcoding time and the predicted transcoding time. r^2 or Pearson Correlation Coefficient square [114] represents the co-relation between the features and the output (transcoding time) with values ranging from 0 to 1. For a good transcoding time prediction model, MAE, RMSE and MAPE should be close to zero and r^2 should be close to 1.

Mathematically,

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5.1}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{5.2}$$

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i} \times 100 \tag{5.3}$$

where n is the number of test records, y_i is the actual output (transcoding time) and $\hat{y}_i$ is the predicted output (predicted transcoding time).

It took 1 hour and 56 minutes for auto ML to train the model. The final model returned by AutoML for predicting the transcoding time is an ensemble of 25 models. An ensemble model incorporates prediction results from multiple models to give a final prediction [115]. Ensemble models provide better prediction accuracy in scenarios where a single model fails to accurately map the input-output relationship and combining predictions from several models helps in boosting the prediction accuracy. The final ensemble model consists of 10 artificial neural networks or ANN's (having different topologies and hyperparameters) and 15 boosted decision trees. ANNs and boosted trees are good at capturing non-linear relations and other proposed models [70, 71, 23, 77] have also employed ANNs for predicting the transcoding time. The model generation is summarized in Table 5.5 (Transcoding time prediction). Vertex AI outputs log messages containing information about the architecture (hyperparameters) of the final model as well as the models used during the testing

Table 5.5: AutoML tuning for predicting the transcodign time and number of CPU cores

Property	Transcoding Time Prediction	CPU Cores Prediction
Type	Regression Model	Classification Model
Dataset Size	30772	12858
Train:Validation:Test Split	80:10:10	80:10:10
Optimization Parameter	Mean Absolute Error	Log Loss (Categorical Cross Entropy)
AutoML Model	Ensemble Model: 10 ANN and 15 Boosted Decision Trees	Ensemble Model: 25 ANN

phase [116]. The hyperparameters logged for the neural network model include Dropout rate, Embedding numerical embedding, enable L1, enable L2, Enable batch Norm, Enable embedding L1, Enable embedding L2, Enable Layer Norm, Hidden Layer Size, Normalize numerical column, Number of cross layers, Number of hidden layers and Skip connection type. The hyperparameters logged for the Decision tree model include Tree complexity, L1 regularization, L2 regularization, Maximum tree depth and Number of trees. The transcoding time prediction model returned by Vertex AI is available at [117] and a summary of the model architecture (major model hyper-parameters) is presented in Appendix (Sec. A.1)

5.3.3 Number of CPU Cores Prediction

We used 11 features to train the number of CPU cores prediction model - duration, fps, input height, output height, input pixels, output pixels, downscaled SI, downscaled TI, downscaled bitrate, preset and the transcoding time. As the transcoding time saturates with increasing cores (Sec. 5.2.5) as evident from Fig. 5.4, the output of the prediction model can map to multiple classes. To overcome this challenge, we decided that for a transcoding task only records till the saturation cores should be included to train the model. For example, for a segment transcoding $T_{s->t}$ where s is the input segment and t is the transcoded segment, if the transcoding time saturates after n cores we discard all transcoding records for $T_{s->t}$ for which CPU cores is greater than n. We define saturation cores as the core value after which

the decrease in transcoding time is less than S percent. We set the value of S to be 8% for our research as it gave a good decision boundary as to when transcoding time speedup stops. We had a total of 12858 records for training the dataset.

We used log-loss as the optimization metric which is the cross entropy between the predicted number of CPU cores and the actual number of CPU cores. Log-loss is the only supported optimization choice for multi class classification in Google AutoML. For evaluating the classification model performance, we computed the accuracy, precision, recall and f1-score. Accuracy as shown in Equation 5.4 is the ratio of correct predictions to the total predictions made. Precision as shown in Equation 5.5 is a ratio of correct predictions (True Positives) for a class to all predictions that are mapped to that class. A high value of precision implies low False Positives for a class. Recall as shown in Equation 5.6 is the ratio of correct predictions (True Positives) made for a class out of all the samples which belong to that class. A high value of recall implies low False Negatives for a class. f1-score as shown in Equation 5.7 is the harmonic mean of precision and recall. Accuracy examines the ability of the model to make correct predictions and is suitable evaluation metric for balanced datasets where the distribution of samples is balanced between different classes. For imbalanced datasets where the distribution on samples among different classes is uneven f1-score outperforms accuracy [118] as it takes into consideration both the False Positives and the False Negatives for a class and ensures that the model is capable of correctly predicting all the classes as opposed to being biased towards some classes.

$$Accuracy = \frac{Correct\ Predictions}{Total\ Number\ of\ Samples\ (Predictions)} \times 100 \tag{5.4}$$

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \times 100 \tag{5.5}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \times 100 \tag{5.6}$$

Table 5.6: Performance Metrics for the transcoding time prediction model

Metric	Value
Mean Absolute Error (MAE)	0.116 seconds
Root Mean Squared Error (RMSE)	0.263 seconds
Mean Absolute Percentage Error (MAPE)	2.191 %
r_squared(r^2)	0.997

$$f1\,score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.7}$$

AutoML trained the CPU cores prediction model for 2 hours and 11 minutes and the final model returned is an ensemble of 25 ANNs. The model is summarized in Table 5.5(CPU cores prediction). The number of CPU cores prediction model returned by Vertex AI is available at [119] and a summary of the model architecture (major model hyper-parameters) is presented in Appendix (Sec. A.1)

5.4 ML Model Evaluation

In this section, we evaluate the performance of the two prediction models. We examine the performance metrics for the two models along with the feature importance/correlation metrics.

5.4.1 Transcoding Time Prediction Model

The regression model returned by AutoML captures the relation between the transcoding features and the transcoding time quite well when applied to the testing dataset as is suggested by the performance metrics presented in Table 5.6. The model has a low MAE value of 0.116 seconds and a low RMSE value of 0.263 seconds indicating that the transcoding time predicted by the model is close to the actual transcoding time. The MAPE for the model is 2.19% which implies that the prediction deviates from the actual value by just a small fraction. Furthermore (r^2) has a value of 0.97 indicating a high-quality model.

Fig. 5.7 shows the importance of each feature in predicting the transcoding time. We observe that output pixel is the most important feature and input pixel the least. Output pixel determines the number of pixels to be encoded per frame and hence is a dominant feature. Preset determines how many CPU cycles to spend on encoding a sequence and hence is also an important prediction feature. Duration and fps determine the number of frames to be processed and hence are important prediction determinants as well. Output height and CPU cores determine the parallelism extent achieved by WPP and hence are important features as well. Output height determines the number of CTU rows in a frame and the higher the number of CTU rows, the higher will be the extent of WPP. The number of CPU cores determines the number of threads which can take part in parallel processing of CTU blocks and thus impact the transcoding time. For complexity-based features, the order of importance is downscaled bitrate followed by downscaled TI and downscaled SI, respectively. CRF encoding adjusts the bitrate to reach a target quality and hence the downscaled bitrate is a good estimator of video complexity whereby hard to encode segments get encoded with more bits requiring more time compared to easy to encode segments. Temporal complexity (TI) is based on computing the motion across frames and therefore to encode a segment with a high TI value is computationally more expensive than a spatially complex segment [23].

5.4.2 Number of CPU Cores Prediction Model

The performance metrics of the classification model for predicting the number of cores for achieving a target transcoding time is presented in Table 5.7. The model has a high accuracy of 98.3% indicating that most of the predictions are mapped to the correct class labels. The model's precision is 99.5% high, suggesting that the proportion of false positives is small compared to true positives. Furthermore, there are only a small number of false negative predictions as indicated by a high recall score of 96.4%. Finally, the f1-score, the harmonic mean of precision and recall, was 0.97.

Fig. 5.8 depicts the confusion matrix for the classification model. From the confusion

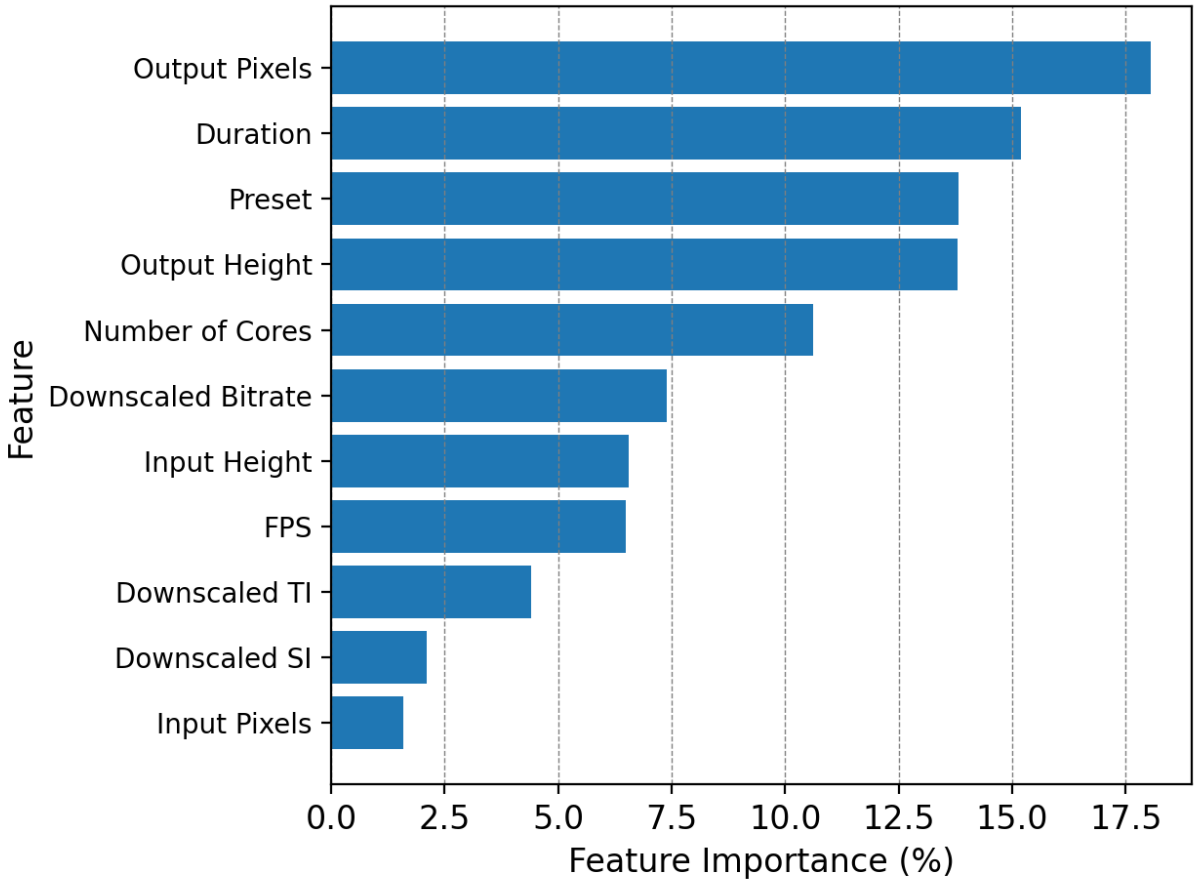


Figure 5.7: Feature importance in predicting transcoding time

Table 5.7: Performance Metrics for the number of CPU cores prediction model

Metric	**Value**
Accuracy	98.3 %
Precision	99.5 %
Recall	96.4 %
F1 Score	0.97

metrics we infer that the model's performs well when the number of cores is less, and the performance of the model drops as the number of cores increases. The proportion of misclassified predictions increases as we move from a small number of cores to a higher number of cores. There are two main reasons for this behaviour. Firstly, the transcoding time difference between consecutive number of cores is more at smaller core values, leading to more defined prediction boundaries between the classes at lower number of cores. For instance, when a 2-second long segment from `Racenight` video was transcoded from 2160p to 1080p with the veryfast preset, the transcoding time difference between 2 cores and 3 cores transcoding was 5.34 seconds where the difference was just 0.31 seconds for 4 cores and 5 cores transcoding. Secondly, the dataset used for generating the model is an imbalanced one as there are more number transcoding records with lower number of cores and a smaller number of transcoding records with higher number of cores (for a segment transcoding we removed all those records for which the allocated cores was higher than the saturation cores). The training phase saw more low core records and was able to correctly map input-output relationship for these low core records. We let the dataset as it is because it represents the distribution of segments from real videos. However, one thing to note is that if the predicted cores are more than the actual cores, the prediction will lead to resource over-provisioning but will not have any negative impact on the transcoder throughput as the throughput always increases (or stays constant) with increasing CPU cores. Overall the number of CPU cores prediction modal performed well.

Fig. 5.9 shows the impact each feature has in predicting the CPU cores. We observe that transcoding time has the highest impact on the prediction. This is because the less the transcoding time the more the resources the transcoder needs to reach a high transcoding throughput and vice versa. Output Height also has a high impact on predicting the transcoding time since the cores taking part in WPP are dependent on the number of CTU rows in a frame which is a function of output height. Preset is also a crucial factor as it governs the transcoder speed and the transcoding bitrate/quality. For complexity-based features

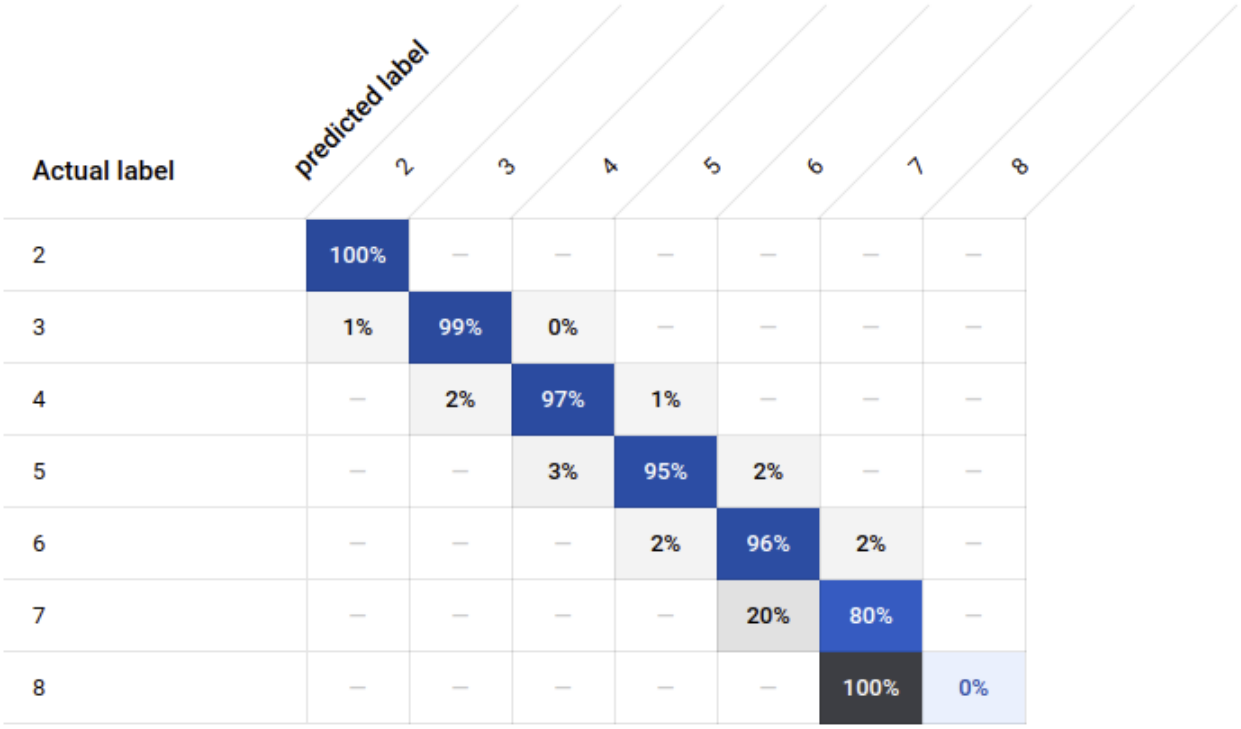


Figure 5.8: Confusion matrix for the number of CPU cores on the test dataset

downscaled bitrate is the most important prediction feature.

5.5 Vertex AI comments

Vertex AI largely simplifies the development of a ML pipeline. No libraries, dependencies need to be installed by the user and all the training and testing takes place in the cloud. Vertex AI takes care of data pre-processing and ensures that unrelated features do not pollute the final predicted values. Additionally Vertex AI returns the final model (along with the model architecture logs) which can be deployed to make real-time predictions.

Although Vertex AI makes the model training job easy, there are certain drawbacks. Firstly, since Vertex AI tries to fit many different models (models with different topologies and hyperparameters) to the transcoding dataset, the training process takes a long time. Specifically, it took 1 hour 56 minutes to train the transcoding time prediction model and 2 hours 11 minutes to train the CPU cores prediction model. Secondly, although Vertex AI

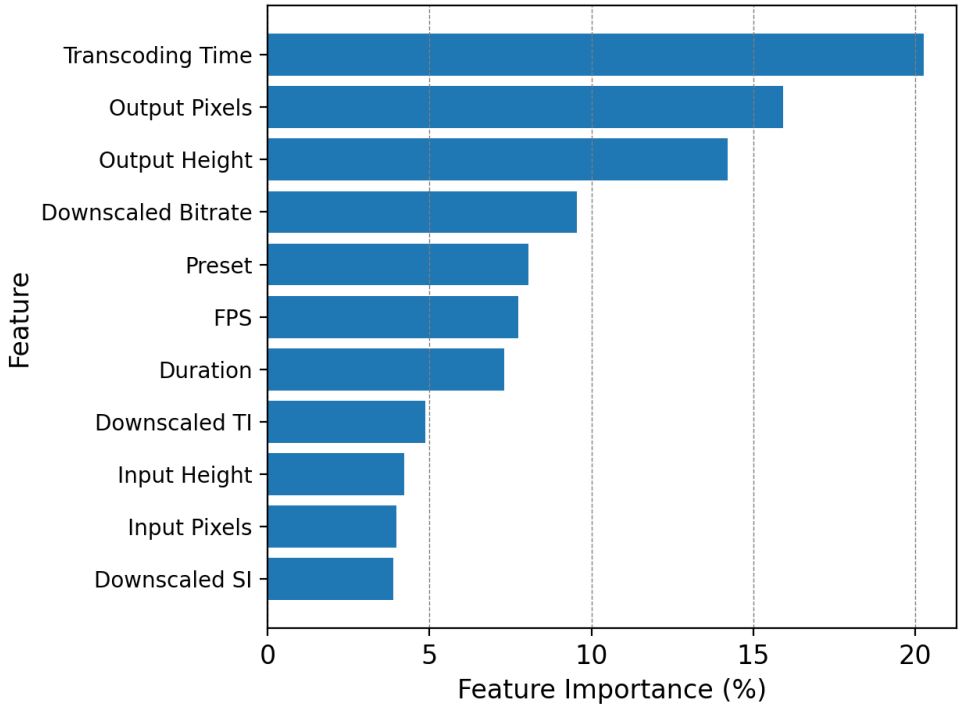


Figure 5.9: Feature importance in predicting the number of CPU cores

gives the architecture of the different models (hyperparameters) it tried during the training stage as well as the architecture of the final model, it is still a challenge to explain the decision choices made by Vertex AI in coming up with the final model. Thirdly, the model returned by Vertex AI for the predictions consists of multiple models which makes the overall prediction pipeline heavy.

Despite these limitations, we used Vertex AI, as it allowed us to train the prediction models with ease. We simply gave the transcoding dataset without any modifications (pre-processing) to Vertex AI and it took care of the model generation. Vertex AI is also smart enough to discard features which are not correlated to the target variable. This is one of the main advantage of using Vertex AI as we don't have to run a sensitivity analysis to evaluate the importance of features and vertex AI automatically does that for us (Feature importance matrix). Also, to make the model evaluation faster Vertex AI uses dropout rates (dropping some of the neurons) for the neural networks so that even with multiple models,

the computational complexity and the execution time of the overall prediction model is reduced.

The features we selected for training the prediction model are mostly inspired by prior works published in literature. However, some of the features are correlated and thus may be redundant. For instance, input pixels is correlated to the input height (resolution) and similarly output pixels is correlated to the output height. Also, from the feature correlation metrics (Fig. 5.7, Fig. 5.9) we observe that some of the features contribute more towards the final predictions. Therefore, reducing the feature space (to less than 11 features) to further optimize the prediction model is worth investigating.

Since we selected the random train:test split when generating the AutoML models, it is not possible to exactly replicate/reproduce our Google AutoML models.

5.6 Balancing Performance and Resource Efficiency

The ultimate goal of our predictions is to determine the minimum number of cores needed to either maximize the transcoder throughput (minimize the transcoding delay) or to at least make the transcoder throughput meet the frame rate of the streamed video to provide a smooth interruption free playback. When a streaming session begins the segments should be transcoded as fast as possible so that the player buffer can be filled faster allowing the client to start the playback quickly. During this initial delay phase (*scenario 1*), we need to determine the minimum number of cores needed to achieve the fastest transcoding rate. Once the playback begins, the buffer gets depleted as segments are removed from it, and new segments must fill the buffer at the same rate (i.e playback rate) to keep the buffer level constant. For videos with 1-sec segments, the rate should be one segment every second. For streams with 2-sec segments, the rate should be one segment every two seconds. Thus, under stable condition (*scenario 2*), the transcoder should transcode one segment every 1 or 2 seconds (depending on the segment duration). We need to determine the minimum number

of cores needed to achieve this rate. In fact, under stable conditions, the transcoding does not have to be faster than this rate because the live source would be producing segments at this rate as well.

We combined two prediction models from Sec. 5.3.2 and Sec. 5.3.3 to determine the optimal resource allocation for each of the two scenarios.

1. *Scenario 1 – Fastest transcoding rate:* We query the transcoding time prediction model to get the transcoding time when N cores are allocated to the container. We begin with the largest number of cores supported by our dataset as the N value and decrease N until we notice an increase in transcoding time. From this process, we find the fastest transcoding time t and the minimum number of cores needed N^{min} to achieve this transcoding time.

2. *Scenario 2 – Meeting the live rate:* We query the CPU cores prediction model with the target transcoding time T (which is also the segment duration) to obtain the minimum number of cores needed N^t (However, if the target transcoding time is less than what what the system can do at best, the number of cores returned by the model will be for the lowest transcoding time possible on the system).If the target transcoding time cannot be met with N^t cores (we query the transcoding time prediction model with N^t cores to determine this), we run the algorithm for scenario 1 to get the fastest transcoding time t and the corresponding number of cores N^{min}. The transcoder should then schedule $\lceil \frac{t}{T} \rceil$ segments to be transcoded in parallel to reach the target transcoding throughput. The degree of parallelism is the number of segments that should be transcoded in parallel to ensure that the overall transcoding rate meets the playback rate.

In a real live streaming session, new segments may fill the playback buffer at a rate slower than the playback rate due to bandwidth fluctuation, which can result in a drop in the buffer level. The buffer could potentially be completely emptied out leading to a playback stall. To

refill the buffer, the client may request new segments at lower quality levels or may request multiple segments at the same time depending on the streaming protocol in use. This may require our resource allocation algorithm to switch back and forth between scenario 1 and scenario 2 throughout a streaming session to meet the needs of the client.

We evaluated the proposed algorithm for balancing the performance and resource efficiency for real-time transcoding. We randomly selected 200 transcoding records from our transcoding dataset and applied the algorithm discussed above (*Scenario 1 and Scenario 2*) to see how the proposed algorithm manages to transcode a live stream with minimum resources. The adaptations in *Scenario 1 - Fastest transcoding rate* as well as in *Scenario 2 - Meeting the live rate* are presented in the following subsections.

5.6.1 Scenario 1: Fastest Transcoding Rate

We compared our proposed algorithm with a baseline algorithm which always allocates fixed number of cores for a transcoding task solely based on the source and target resolutions (as was suggested in Contrast [38]). Specifically, the baseline algorithm allocates 8 cores for 2160p $\rightarrow$ 2160p transcodings, 5 cores for 2160p $\rightarrow$ 1080p transcodings, 4 cores for 2160p $\rightarrow$ 720p transcodings, 4 cores for 2160p $\rightarrow$ 480p transcodings, 5 cores for 1080p $\rightarrow$ 1080p transcodings, 4 cores for 1080p $\rightarrow$ 720p transcodings and 3 cores for 1080p $\rightarrow$ 480p transcodings. We compare the baseline algorithm and our prediction in Table 5.8. We conclude that (1) Overall, our prediction algorithm allocates a smaller number of cores than the baseline algorithm does; (2) The fastest transcoding time achieved by both algorithms are very close (around 1% difference); and (3) With the ML-base system, more resources are conserved for higher target resolutions. The resource savings (CPU core savings) is the highest for 2160p $\rightarrow$ 2160p conversion but this comes at the cost of algorithms transcoding time being 4.25% higher than baseline transcoding time. This happens because of the accuracy of the prediction drops for higher target resolution transcodings which need more cores as discussed in Sec. 5.4.2.

Table 5.8: Scenario 1: Comparing the baseline algorithm and proposed ML based algorithm in terms of transcoding time and resources needed

Conversion	Baseline Resource allocation Strategy		Proposed ML based Resource allocation Strategy		Time Difference	Core Savings
	Number of Cores (Fixed)	Transcoding Time (Seconds)	Average Number of Cores (Dynamic)	Transcoding Time (Seconds)		
2160p - 2160p	8	6.90	6.30	7.20	4.25%	21.25%
2160p - 1080p	5	4.38	4.25	4.47	1.90%	15.00%
2160p - 720p	4	4.03	3.85	4.04	0.43%	3.67%
2160p - 480p	4	2.88	3.96	2.88	0.08%	0.83%
1080p - 1080p	5	3.43	4.62	3.41	0.72%	7.50%
1080p - 720p	4	2.86	3.85	2.86	0.12%	3.57%
1080p - 480p	3	2.51	2.97	2.49	0.84%	0.90%
Average	**4.51**	**3.71**	**4.12**	**3.75**	**1.07%**	**8.6%**

5.6.2 Scenario 2: Meeting the Live Rate

For the 200 randomly chosen transcoding records the number of segments needed to be transcoded in parallel to match the video frame rate are given in Table 5.9. The higher the target resolution, the more the segments that need to be transcoded in parallel to meet the stream frame rate. For most cases, the degree of parallelism is between 2 and 3, the impact of which can be masked by the playback buffer on the client side.

Table 5.9: Number of Parallel transcoded segments needed to sustain the live streaming rate

Conversion	**Number of Parallel segments to be transcoded for 1-second long segments**	**Number of Parallel segments to be transcoded for 2-second long segments**
2160p - 2160p	6.5	5
2160p - 1080p	4.38	3.66
2160p - 720p	3.73	3.13
2160p - 480p	3.21	2.18
1080p - 1080p	3.77	2.66
1080p - 720p	3.09	2.42
1080p - 480p	2.33	2.06

5.7 Conclusions

In this chapter, we proposed a novel machine learning based solution to predict the resources required to achieve a target transcoding throughput. We first crafted a transcoding dataset and then developed 2 ML models. The first model predicts the transcoding time and the second one predicts the resources needed. Evaluation results show that our model can predict transcoding time with a MAPE of 2.191% and CPU cores with an accuracy of 98.3%. Furthermore, we compared our proposed Machine Learning based resource allocation strategy to a baseline fixed resources allocation strategy and observed that the proposed ML strategy can lead to resource savings.

Chapter 6

Antifreeze: Adaptive Real-time Transcoding and Streaming

In a live streaming setup, the server transcodes live segments on-the-fly in real-time as they become available. With real-time transcoding, it is possible to dynamically adapt to the client's needs and playback conditions during run-time, thus offering the viewer a more personalized streaming experience. Real-time transcoding could potentially lead to a high-quality, smooth stream and efficient bandwidth utilization. However, for dynamically transcoding a live stream in real-time, adaptation rules that consider the on-the-fly transcoding overheads must be defined. In this chapter, we propose `Antifreeze`, an adaptive real-time transcoding, and streaming system.

The rest of this chapter is organized as follows. In Sec. 6.1, we present the buffer model for a live stream transcoded in real-time on an on-demand basis. Then in Sec. 6.2 we present a loss function that considers the transcoding time, bandwidth, buffer level, and visual quality to determine the best quality on-the-fly transcoded variant that can maximize clients QoE without risking playback stalls. Finally, in Sec. 6.3, we present the design of `Antifreeze`. The implementation and evaluation of `Antifreeze` is presented in Chapter 7

6.1 Client Buffer Model

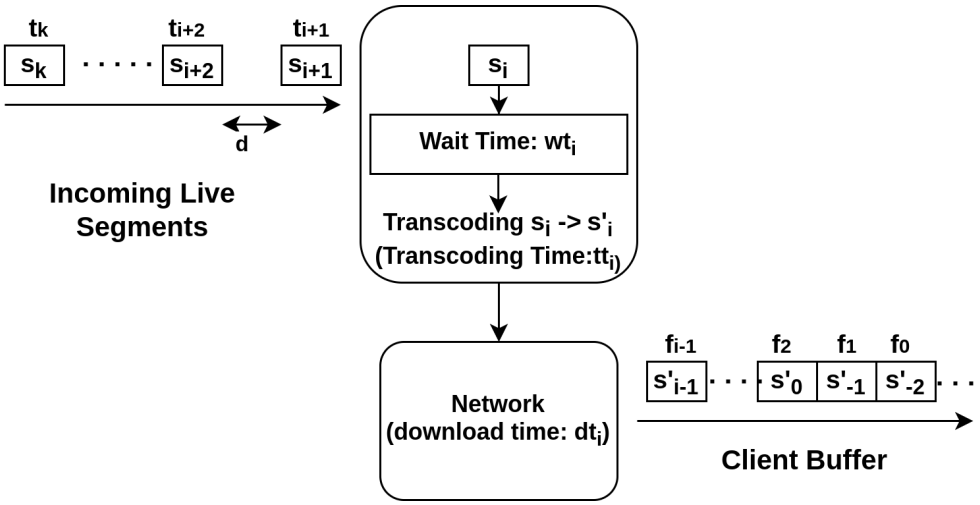


Figure 6.1: Real-time transcoding architecture depicting the live source, transcoding server, network and the client buffer

Let $\{...s_{-2}, s_{-1}, s_0, s_1, s_2, s_3 \dots\}$ denote the set of live streamed segments such that s_i is generated at time instant t_i and the duration of each segment is `d` seconds, as shown in Fig. 6.1. Since consecutive segments in a live stream are generated periodically after a delay of segment duration `d`, we have

$$t_1 - t_0 = t_2 - t_1 = t_3 - t_2 = \ldots = \mathtt{d}$$

or

$$t_{i+1} - t_i = \mathtt{d} \tag{6.1}$$

When the live streaming session begins, the client starts fetching the video segments from the server and stores them in its buffer in sequential order. This initial buffer-building phase lasts till the client has stored segments till s_{-1} into its buffer. After the next segment s_0 has been downloaded at f_0, playback begins. Without loss of generality, our evaluation here focus on the steady state analysis of the buffer (*i.e.,* after the initial buffer building phase) after playback has started. The buffer level after receiving the first segment s_0 at the

beginning of the steady state (f_0) is given by:

$$l_0 = \mathtt{l} + \mathtt{d} \tag{6.2}$$

where `l` is the instantaneous buffer level after the initial phase and `d` is the segment duration. This means that the buffer level is increment by a segment duration after receiving this first steady state segment s_0. For each of the subsequent segments s_i, upon receiving the corresponding request, the server transcodes s_i to s'_i (the segment in the appropriate quality level determined by our loss function described in Sec. 6.2) in tt_i seconds. The transcoded segment s'_i is then delivered to the client in dt_i seconds The abbreviations tt and dt stand for transcoding time and download time, respectively. On the client side, the transmission of the segment s_i finishes at time f_i. The buffer level after receiving each segment s'_i is given by:

$$l_i = l_{i-1} + \mathtt{d} - (f_i - f_{i-1}) \tag{6.3}$$

where l_i and l_{i-1} is the buffer level after receiving segments s_i and s_{i-1}, respectively. The buffer level is updated by adding the duration of the downloaded segment s_i (denoted by `d`) and then subtracting the playback duration. Since smooth playback means continuous playback without stalls (buffer never gets empty) as time goes on, the playback duration while waiting for segment s_i is from the time that the client requests segment s_i from the server (which is right after receiving segment s_{i-1}) at time f_{i-1} to the time that the client received segment s_i at time f_i, *i.e.,* $f_i - f_{i-1}$. The buffer level change during this playback interval is calculated as follows:

$$\Delta l_i = l_i - l_{i-1} \tag{6.4}$$

By substituting Eqn. 6.3 into Eqn. 6.4, we get:

$$\Delta l_i = \mathtt{d} - (f_i - f_{i-1}) \tag{6.5}$$

In the end-to-end streaming system, the life-cycle of each segment s_i starts at time t_i (when it was generated by the source), followed by wait time wt_i on the server waiting for the client to send a request for it. Upon receiving the request, it takes the server tt_i to transcode it to the desired quality level, and then dt_i to deliver it to the client. Thus, f_i and f_{i-1} can be defined as follow:

$$f_i = t_i + wt_i + tt_i + dt_i \tag{6.6}$$

$$f_{i-1} = t_{i-1} + wt_{i-1} + tt_{i-1} + dt_{i-1} \tag{6.7}$$

Substituting Eqn. 6.6 and Eqn. 6.7 into Eqn. 6.5, we get:

$$\begin{aligned} \Delta l_i = \mathtt{d} - [(t_i - t_{i-1}) + (wt_i - wt_{i-1}) \\ +(tt_i - tt_{i-1}) + (dt_i - dt_{i-1})]) \end{aligned} \tag{6.8}$$

Now, substituting Eqn. 6.1 and simplifying, we get:

$$\Delta l = -(\Delta tt + \Delta dt + \Delta wt) \tag{6.9}$$

The buffer change from receiving one segment to the next is related to the transcoding time (tt), the download time (dt), and the wait time (wt). The transcoding time depends on the complexity of the segment and the underlying hardware. The download time depends on the segment size and available bandwidth. The wait time varies depending on when the segment becomes available (when the segment is generated by the live source) and when the client requests it. As a segment becomes available, it is either served immediately if the client has already requested it (*i.e.,* $wt_i = 0$ when *i.e.,* $f_{i-1} \leq t_i$), or wait to be served until the client requests it. In the latter case, the wait time is the difference between the time the request was sent at time f_{i-1} and the time this segment become available at time t_i. Thus, the wait time can be expressed as follow:

$$wt_i = \begin{cases} 0, & \text{if } f_{i-1} \leq t_i \\ f_{i-1} - t_i, & \text{otherwise} \end{cases}$$

This can be further simplified to $wt_i = \max[0, f_{i-1} - t_i]$. By substituting Eqn. 6.7, we get:

$$wt_i = \max[0, (wt_{i-1} + tt_{i-1} + dt_{i-1} - d)] \tag{6.10}$$

6.2 QOE Maximizing Loss Function

The client requests for segment s_i at a target quality of q_i. The server transcodes s_i to s'_i where the quality level of s'_i is q'_i and delivers s'_i to the client. In an ideal real-time transcoded streaming session, we aim to minimize the following losses

1. Buffer loss ΔB: Maintaining a healthy buffer level, *i.e.,* $\Delta l_i >= 0$. This loss is proportional to the amount by which buffer dropped ($\Delta l_i < 0$). Thus, we only count the buffer drop by taking $min(0, \Delta l_i)$.

2. Quality loss ΔQ: Providing the segment close to the requested quality, *i.e.,* $q'_i \approx q_i$.

3. Quality switch loss ΔS: Maintaining consistent quality from segment to segment, *i.e.,* $q'_i \approx q'_{i-1}$.

However, getting an ideal transcoded streaming session is challenging due to the bandwidth constraints and the transcoding overheads. In this section, we formulate a decision model capable of delivering s_i and maintaining the target quality level q_i while minimizing playback stalls. The model takes in a target quality level (q_i) along with other streaming parameters as inputs and determines the most optimal quality level to which a requested segment should be transcoded by the server.

To determine the best quality variant to which a segment s_i should be transcoded on-the-fly, we introduce a loss function $L(s'_i)$ that determines how close the transcoded segment s'_i is to an ideal transcoded segment meeting the three QoE goals stated above. The loss function can be formally expressed as follow:

$$L(s'_i) = \Delta B + \Delta Q + \Delta S$$

More specifically:

$$L(s'_i) = a[\min(0, \Delta l_i)]^2 + b[q_i - q'_i]^2 + c[q'_i - q'_{i-1}]^2 \tag{6.11}$$

The loss function $L(s'_t)$ for a transcoded segment s'_i is an aggregate of three losses: buffer loss ΔB, quality loss ΔQ, and quality switch loss ΔS. The coefficients a, b and c are weights assigned to each of the three losses. We take the square values of each term to make the values positive so that we can minimize the overall loss.

Quality loss and quality switch loss were proposed by Qin *et al.* in their design of QUAD [16]. The formulation of buffer loss which also considers the transcoding time is a major contribution of this thesis.

6.3 Antifreeze

Antifreeze is an adaptive, real-time end-to-end transcoding platform for high-quality live video streaming. The flowchart of Antifreeze is presented in Fig 2. The system consists of a HAS server and a HAS client. When the client needs a new video segment, it sends a request to the server. The server transcodes the requested segment in real-time to the most optimal quality variant based on the client's playback needs and delivers the transcoded segment to the client. The client then updates its buffer and requests for the next segment from the server. The HAS client and the HAS server are described in detail in the following

subsections.

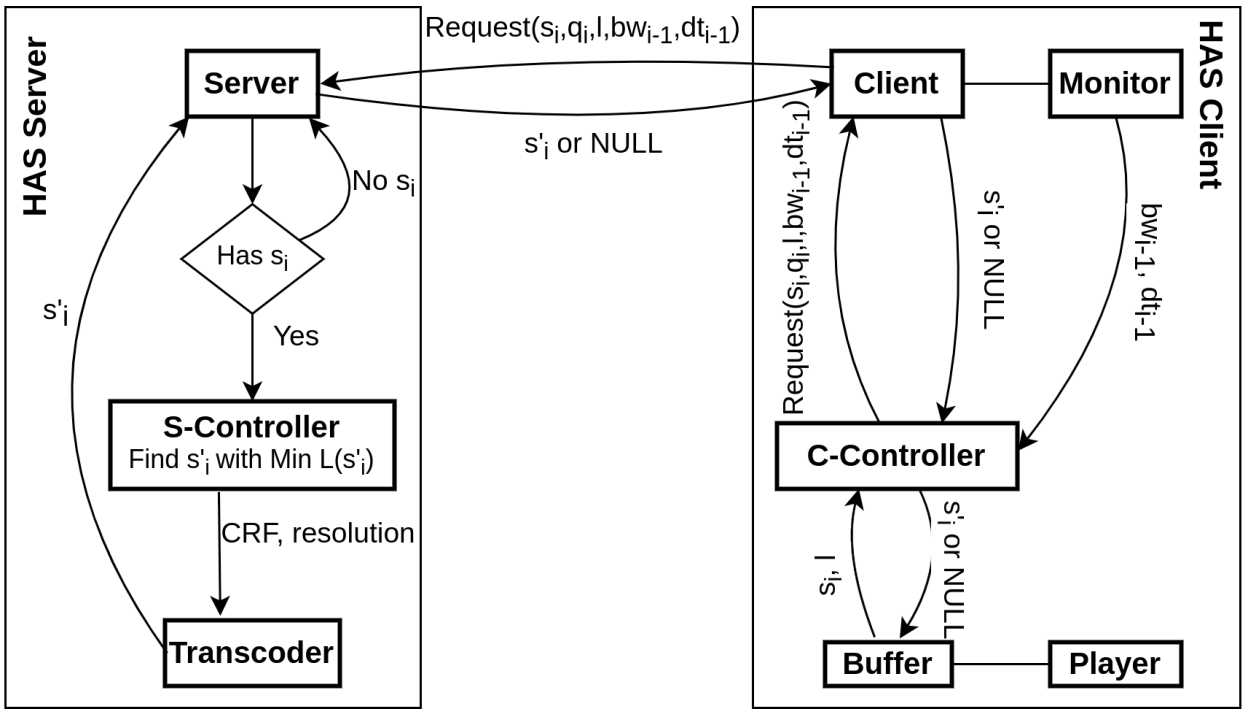


Figure 6.2: Antifreeze Flowchart

6.3.1 Antifreeze Client

When the live streaming session starts, the client requests the first segment from the server and stores the downloaded segment in its buffer and then requests the next segment. With this process of downloading and storing segments, the buffer level rises. The maximum rate at which the buffer can be filled up is limited by the rate at which new segments are generated by the live source. Once sufficient segments have been stored and the buffer level exceeds the threshold level l_{safe}, the client starts dequeuing segments from the buffer, and playback starts [2].

Antifreeze client consist of a Client module, a Client Controller (C-Controller) module, a Buffer module, a Monitor module, and a Player module. The Client module is responsible for sending segment requests to the server and fetching the response from the server. The buffer module keeps track of the current buffer level (the duration of available playback) along

with the segment index s_i that needs to be fetched next. The Player module continuously monitors the buffer and is either in a playing state or a buffering state. The Monitor module records the time it took for a segment to be downloaded and it recomputes the bandwidth after each download. The bandwidth can be estimated by either using the past segment throughput (instantaneous bandwidth) or by averaging out the throughput of the past segments (smoothed average method or harmonic mean method).

The C-Controller module controls the entire cycle of requesting segments from the server and handling the response from the server by acting as an interface between the Client module, the Buffer module and the Monitor module. To issue a new request to the server, the C-Controller queries the buffer module, and the monitor module to get the required playback parameters. Specifically, the C-controller fetches the segment index s_i and the instantaneous buffer level ℓ from the buffer module and fetches the download time of the last downloaded segment dt_{i-1} and the last estimated bandwidth bw_{i-1} from the monitor module. Additionally, the C-Controller also determines the quality level q_i (user requested quality) of the segment it wants to fetch from the server. The C-Controller, after assembling all the request parameters, passes the request to the client and the client then sends the request to the server. The request sent to the server contains the segment index s_i which the client wants to fetch along with the expected quality level q_i, instantaneous buffer level ℓ, the last estimated bandwidth bw_{i-1} and the download time of the last fetched segment dt_{i-1}. The response from the server is either the transcoded segment or a NULL response. NULL response indicates that the server does not have the segment yet and the client must wait for the requested segment to be generated by the live source before it can be transcoded and delivered. The response is passed on to the C-Controller and the C-Controller then updates the buffer. If the response was a transcoded segment, then the transcoded segment is pushed to the buffer, and the segment index is incremented by one (s_{i++}). For NULL response the segment index is not incremented indicating that the client will retry fetching segment s_i until the server has it. The C-controller then repeats the process of querying the

buffer module and the monitor module and issues a new request to the server.

6.3.2 Antifreeze Server

Antifreeze server transcodes the live segments in real-time to the most optimal quality variant based on the client's needs and delivers the transcoded segment to the client. Antifreeze Server consists of a Server module, a Server Controller (S-Controller) module, and a Transcoder module. During the streaming session when a client requests a segment s_i from the server, two cases can arise:

- The server does not have s_i yet (s_i has not been generated) and the server sends a NULL response to the client.
- The server has s_i and it transcodes s_i to the most optimal quality variant and sends it over to the client.

To determine the most optimal variant to which a segment should be transcoded in real time the server queries the Server-Controller (S-Controller) module. The S-Controller module contains a set of adaptation rules based on the loss function described in Sec. 6.2 and outputs a CRF value and a resolution value as target transcoding parameters based on the client's needs and the playback conditions. The computed CRF and resolution values are then passed as parameters to the transcoder module which in real-time transcodes the segment under these parameter settings and the transcoded segment is then delivered to the client by the Server module. The S-controller is described in detail in the next subsection.

6.3.3 S-Controller

As previously stated, when a client requests for a segment s_i from the server, it sends the instantaneous buffer level ℓ, last estimated bandwidth bw_{i-1}, target quality q_i, and the download time of the previously fetched segment dt_{i-1} as client-side playback parameters to the server. The S-Controller then dynamically computes the most optimal transcoding

parameters (CRF and resolution) under which the requested segment must be transcoded in real-time such that the transcoding does not risk the client buffer level dropping below the safe buffer level l_{safe} while ensuring that the stream quality is consistent and close to the user requested quality. An effective strategy to ensure that the client buffer stays above the safe buffer is to make the rate at which the client buffer is being emptied out due to playback match the rate at which new content is being fetched from the server. For segments d seconds long, this implies that each segment must be transcoded, and delivered to the client in a time less than or equal to d seconds. However, segments vary in complexity even when belonging to the same video stream and transcoding the segment under the same timing constraints is thus sub-optimal. Complex segments require higher transcoding time and higher bitrate to reach a target quality than simple segments. Allowing transcoding flexibility based on the segment complexity can enhance the QOE and S-Controller allows for that flexibility. S-Controller permits hard to encode sequences to be transcoded and downloaded in times higher than d seconds. Although this will result in buffer depletion at the client end and a segment queue buildup at the server (because the video transcoding and download rate drops below the live video rate), but S-Controller is optimized to bring the client buffer near safe levels again by transcoding and delivering the queued-up segments quickly under d seconds each resulting in a buffer recovery.

The design of the S-Controller is illustrated in Fig. 6.3. To compute the most optimal transcoding parameters (*i.e.,* CRF and resolution), the S-Controller computes the QoE loss function for each possible combination of CRF and resolution values for which the transcoded segment quality is equal to or less than the user-requested quality q_i. Quality loss and Quality Switch loss can be computed by comparing the quality of the transcoded segment q_i' with the user requested quality q_i and previously streamed segment quality q_{i-1}' respectively (as suggested by [16]). We optimized the buffer loss component described in Eqn. 6.11 so that the buffer loss not only depends on the buffer change Δl_i but also on the instantaneous buffer level ℓ. The Inclusion of instantaneous buffer value allows the buffer to recover back to a

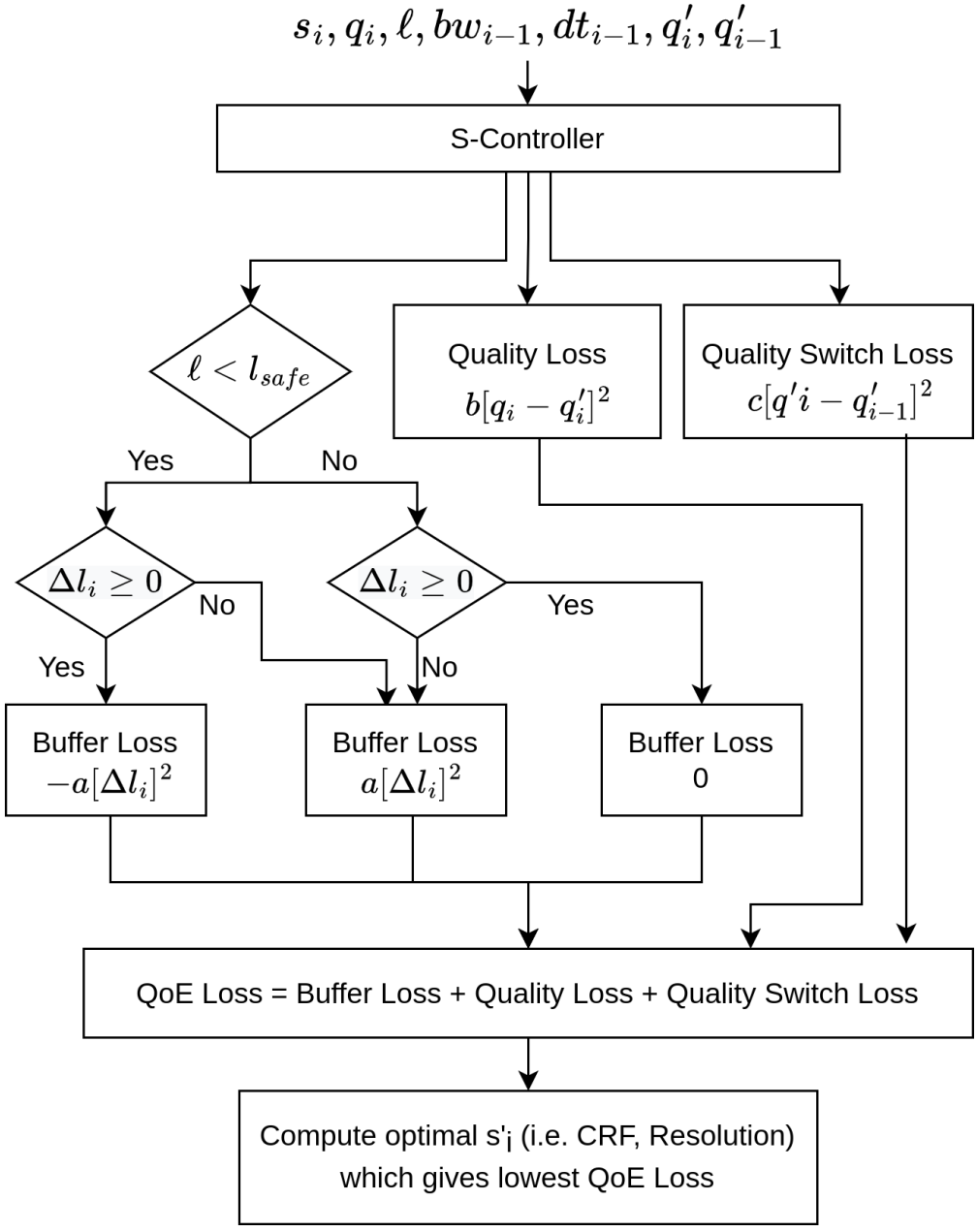


Figure 6.3: S-Controller Flowchart

safe level (l_{safe}) in cases when the buffer has dropped either due to bandwidth drops or due to transcoding overheads. The modified buffer loss component is structured as follows:

- If $\ell < l_{safe}$ and $\Delta l_i < 0$: A buffer loss proportional to the buffer change is applied since the buffer will drop further below the safe level. Such a transcoded variant is unfavorable as it can lead to stalls.

- If $\ell < l_{safe}$ and $\Delta l_i > 0$: A negative buffer loss proportional to the buffer change is applied since the buffer level will rise towards the safe buffer level. Such a transcoded variant is favorable as it will recover the buffer

- If $\ell > l_{safe}$ and $\Delta l_i < 0$: A buffer loss proportional to the buffer change is applied since the buffer level will drop.

- If $\ell > l_{safe}$ and $\Delta l_i > 0$: Zero buffer loss is applied since the buffer is already above the safe buffer level.

Since Quality loss and Quality Switch loss are computed in a quality unit (*i.e.,* VMAF) and Buffer loss is computed in a time unit (*i.e.,* seconds) we normalize the loss function components (as suggested by [16]) before adding them up to get the QOE loss function. We divided the buffer loss component with Safe Buffer level(l_{safe}) and divided the quality loss and quality switch loss component with user requested VMAF quality (q_i) to normalize the components and make them unitless. After evaluating the loss function for all possible CRF-resolution pairs, the S-controller outputs the CRF-resolution pair which has the lowest loss. The server then instructs the transcoder to transcode the requested segment to the computed resolution with the computed CRF value to generate the optimal quality transcoded variant in real-time.

In the next chapter, we describe the implementation details of `Antifreeze` along with its performance evaluation.

Chapter 7

Evaluation of Antifreeze

In Chapter 6, we introduced Antifreeze, an adaptive real-time transcoding and streaming system. In particular, we modeled the client-side buffer for a real-time transcoded stream. Using the client buffer model we proposed a loss function, which represents the deviation of a transcoded segment from an ideal segment where an ideal segment is the one that offers the highest possible QoE to the end user in terms of stream smoothness, stream quality, and consistency in the stream quality. We further proposed `Antifreeze` from the ground-up. Specifically for a transcoding task $s \rightarrow s'$ where s is the source segment and s' is the transcoded segment, `Antifreeze` determines that s' whose quality is close to the user requested quality and which can be generated and delivered to the client fast enough to avoid playback stalls.

In this chapter, we evaluate the performance of `Antifreeze`. We first provide the implementation details of `Antifreeze` in Sec. 7.1. We evaluate the behavior of Antifreeze by investigating the effects of the coefficients a, b, and c under a constant bandwidth connection in Sec. 7.2. Then in Sec. 7.3, we compared `Antifreeze` with two other adaptation algorithms - one optimized for stall minimization and the other one optimized for quality maximization. Finally, to examine the practicality of antifreeze in a real-world deployment, we deployed and evaluated the performance of `Antifreeze` in a variable bandwidth profile in Sec. 7.4.

7.1 Implementation

As outlined earlier, the objective of this research is to propose a QoE aware adaptation algorithm which considers the transcoding time, download time and quality level when deciding on the best fit on-the-fly transcoding variant that should be delivered to the client in real-time. The transcoding time depends on the complexity of the segment and the underlying hardware. The download time depends on the segment size and the available bandwidth. The visual quality depends on the segment complexity and the encoding mode. Therefore, being able to predict the transcoding performance (*i.e.,* transcoding time, bitrate, visual quality) corresponding to the transcoding parameters (*i.e.,* CRF, resolution) is crucial to make antifreeze lightweight and easy to deploy. There have been many models proposed in prior works to predict the transcoding performance or the transcoding parameters. Models to predict the transcoding time include [70, 71, 72, 73, 75, 23, 77, 79, 80]. Models that relate bitrate and CRF include [98, 120, 121, 122]. Models that relate CRF and VMAF include [123, 124]. Many of these proposed models do not have a high accuracy. For instance, the model proposed by [123] to predict the CRF value to reach a target quality had an accuracy of 77.6 %.

Nonetheless, existing prediction models validate the practicality of Antifreeze. However, Antifreeze is about optimizing the transcoding and streaming process to get maximized QoE. In our evaluation, we do not want the prediction inaccuracies to pollute the experimental results. Therefore, to ensure that the prediction inaccuracies does not mislead the end-to-end system, we pre-transcoded the video segments into a variety of transcoded variants and recorded the transcoding stats in a lookup table. Whenever antifreeze adaptation algorithm (*i.e.,* S-Controller) must decide the next optimal transcoded variant, its fetches the transcoding data from the lookup table, computes the loss function for probable transcoded variants and selects the best quality variant. This ensures that there are no inaccuracies in the adaptation algorithm as input data comes from actual transcoding records (as opposed to being predicted). Integrating prediction algorithms within antifreeze is out of scope for

Table 7.1: Description of the video segments used to generate the lookup table

S.N	Video	Resolution	# Segments	Complexity
1	City	3840x2160	5	Low
2	Race	3840x2160	5	High
3	Flower Focus	3840x2160	5	Low
4	Flower Kids	3840x2160	5	High
5	River	3840x2160	5	High
6	Twilight	3840x2160	5	Low

this thesis and is something to be addressed in future work.

To generate the lookup table, we selected 5 videos having different complexities from the UVG Dataset [105]. These videos vary in complexity and Table 7.1 summarizes the properties of the videos. using `FFmpeg`, we split the videos into 2 second segments. Then using `FFmpeg` and the `libx265` encoder, we transcoded each segment into multiple quality levels. Specifically, to cover the widest possible range of quality levels we pre-transcoded segments to 1920 X 1080, 1280 X 720, and 1024 X 576 resolutions under CRF encoding with CRF values ranging from 18 to 42. For each of the pre-transcoding, we recorded the CRF value, the transcoding time (seconds), the video quality (VMAF) and the bitrate (kbps). Thus, the lookup table contains records about all the possible variants to which a segment can be transcoded on-the-fly when request comes in from the client during the live streaming session. We simulate the transcoding process for each segment by introducing a `wait()` equalling the transcoding time of the selected resolution-CRF combination (found in the lookup table) selected by the S-controller.

The transcoding and streaming server is 16 core machine equipped with 62.4 GiB of memory and running ubuntu 20.04. The client and server are both implemented in JavaScript. When the streaming session starts, the client requests for the first segment. The first segment is delivered at a quality level of $q_{initial}$. In the initial phase, the client is in a buffer building stage and during that period those transcoded variants are delivered to the client that have a transcoding time and download time of less than 2 seconds (*i.e.,* segment duration) but whose quality is closest to the user requested quality (q_i). Playback starts after sufficient

segments have been stored in the buffer and the buffer level exceeds the safe buffer level l_{safe}. Once the playback starts, the client goes into playing state and the antifreeze adaptation algorithm kicks in. After each segment download, the bandwidth is re-estimated using a smooth averaging function as depicted below

$$bw = \alpha(bw_{instantaneous}) + (1 - \alpha)(bw_{prev}) \tag{7.1}$$

where bw is the re-estimated bandwidth, $bw_{instantaneous}$ is the instantaneous bandwidth observed while downloading the current segment and is the ratio of segment size to download time, bw_{prev} is the last estimated bandwidth, and α is the smoothing factor.

For all our experiments, unless stated otherwise, the duration of each segment was set to two seconds (*i.e.,* the live source generates one segment every two seconds), the initial quality level $q_{initial}$ was set to 80, the user requested quality q_i was set to a constant value of 90, the safe buffer level l_{safe} was set to 4 seconds and the α value was set to 0.9.

7.2 Impact of Adaptation Coefficients - a, b, c

In this section we study the behavior of antifreeze by analyzing the quality of the transcoded stream and observing the buffer level as a function of coefficients a, b and c under a constant bandwidth profile. We use 3 bandwidth levels for our evaluations - 2000 kbps (Low bandwidth), 5000 kbps (Mid bandwidth) and 8000 kbps (High bandwidth).

7.2.1 Impact of a

In the absence of quality loss coefficient b ($b = 0$) and quality switch loss coefficient c ($c = 0$), `Antifreeze` delivers that transcoded variant to the client for which the buffer loss ΔB is minimum or ideally zero. In scenarios where multiple variants have the same buffer loss, `Antifreeze` selects the variant whose quality is closest to the user requested quality q_i. Fig. 7.1 shows the buffer behavior of antifreeze under different bandwidths when b and c were

set to zero. From the figure, we conclude that irrespective of the bandwidth, `Antifreeze` delivers each segment in a time less than 2 seconds (*i.e.*, segment duration) which results in a smooth stream with zero playback stalls as the buffer never drops below l_{safe}. At higher bandwidth `Antifreeze` can stream higher bitrate segments as shown in Fig. 7.2 (even when b and c are set to zero) without risking the buffer to drop below l_{safe} level.

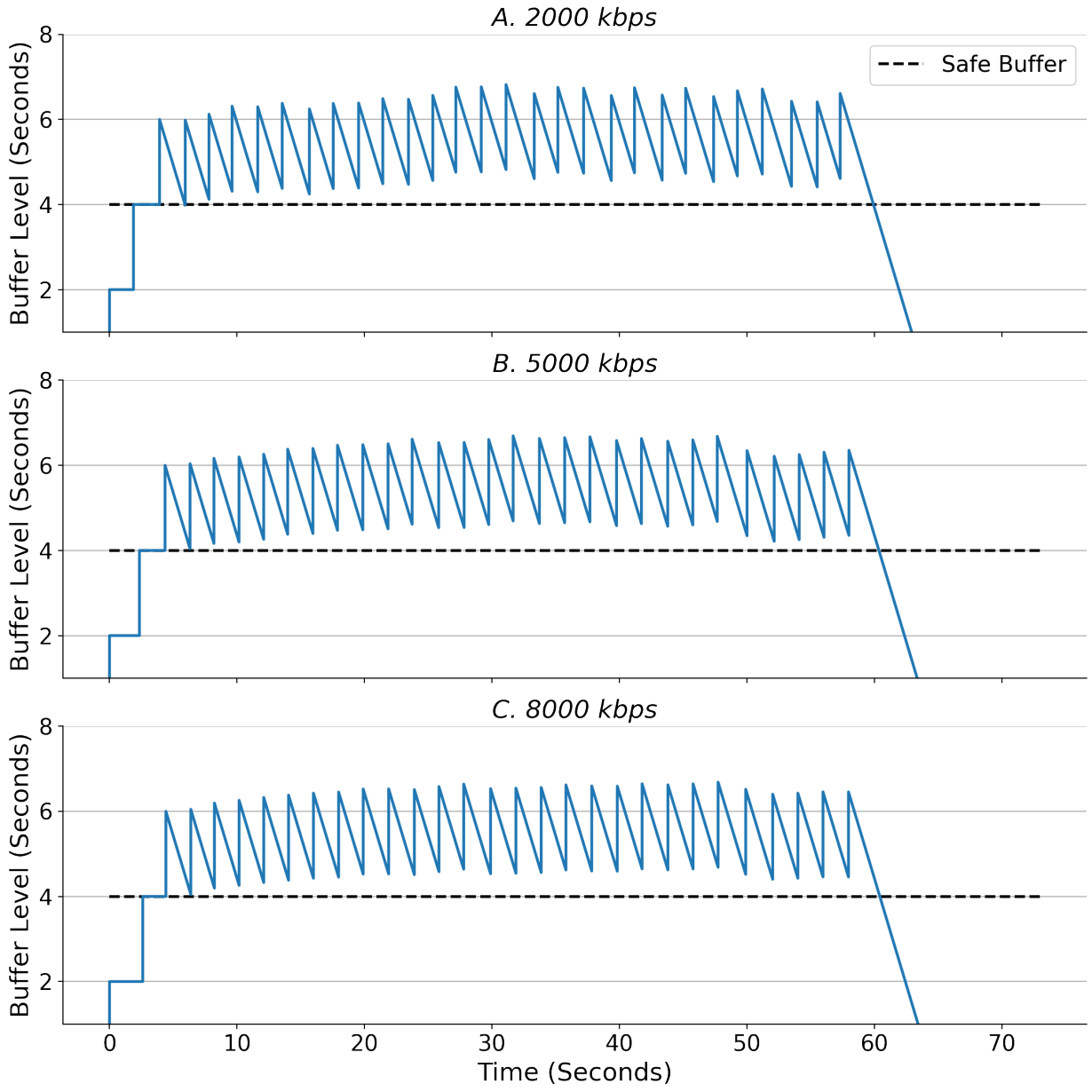


Figure 7.1: Buffer behavior for a=10, b=0 and c=0 under different bandwidth profiles

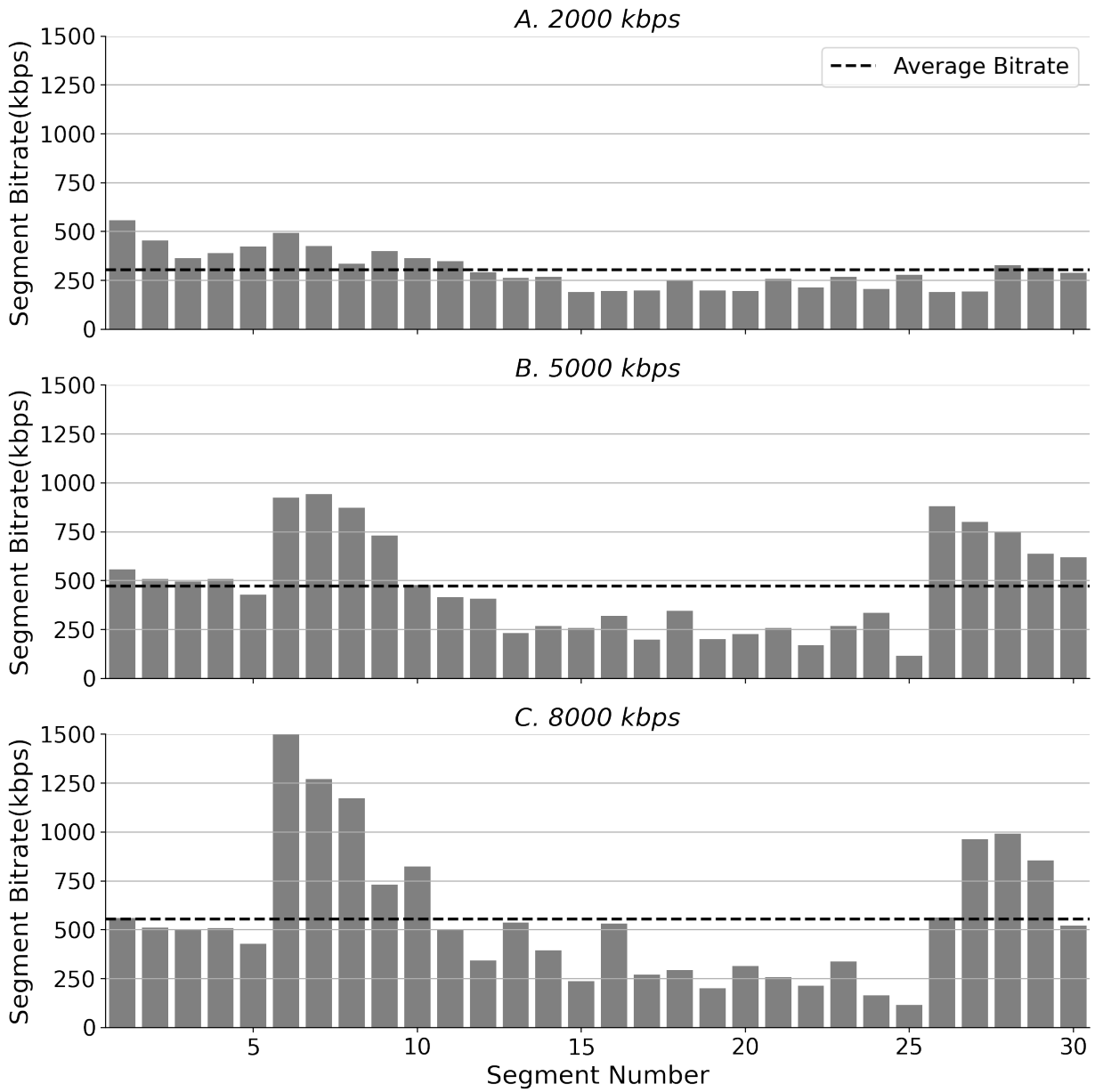


Figure 7.2: Segment wise bitrate distribution for a=10, b=0 and c=0 under different bandwidth profiles

However, on the flip side, in the absence of b, the quality of the stream is poor and much less than the user requested quality level (*i.e.,* 90) as shown in Fig. 7.3(A). The visual quality is much poorer for `Race`, `FlowerKids`, and `River` videos as these video segments have a higher complexity. To enhance the quality of the stream b must be made non-zero. Fig. 7.3(B) shows the improvement in stream quality especially, for complex segments when b was set to 5. In the absence of b, the average VMAF of the stream was just 54.68 which went all the way up to 76.11 when b was set to 5.

With b, `Antifreeze` is adaptive to the segment complexity and allows hard-to-encode sequences to be transcoded and downloaded in a time higher than the segment duration (*i.e.,* 2 seconds). However, if coefficient a is not high enough then b can dictate `Antifreeze` to prioritize quality over transcoding and transmission delays (ΔQ will overpower ΔB) resulting in buffer drops subsequently leading to stalls. Fig. 7.4 compares the buffer behavior as a function of a when b was set to 5 for different bandwidth profiles. When the bandwidth was low (*i.e.,* 2000 kbps), the system was in no position to stream high quality segments and with $a = 5$ and $b = 5$, b essentially dominated the loss function and dictated `Antifreeze` to deliver high-quality segments regardless of the associated transcoding, and transmission delays which resulted in stalls. With increasing a to 10, both the number of stalls and the stall duration reduced as depicted in Fig. 7.4. Stalls specifically appear for sequences that are hard to encode and for sequences that comes after the hard-to-encode sequences as depicted in Fig. 7.5. Hard-to-encode sequences drop the buffer levels which leads to stalls as is the case with `FlowerKids` and `River`. `Twilight` sequences although easy to encode still resulted in stalls as the buffer was simply too low (0 or close to 0) when the twilight sequences were streamed. One more observation we can draw from Fig. 7.4 is the impact a has in bringing the buffer to safe levels again when the buffer has dropped due to a hard to encode sequence. If we observe Fig. 7.4(B). where the bandwidth was 5000 kbps, we can conclude that after the hard to encode sequences (`FlowerKids`, `River`) which resulted in buffer drop were delivered to the client (around 55 seconds time in the buffer timeline), coefficient a started bringing

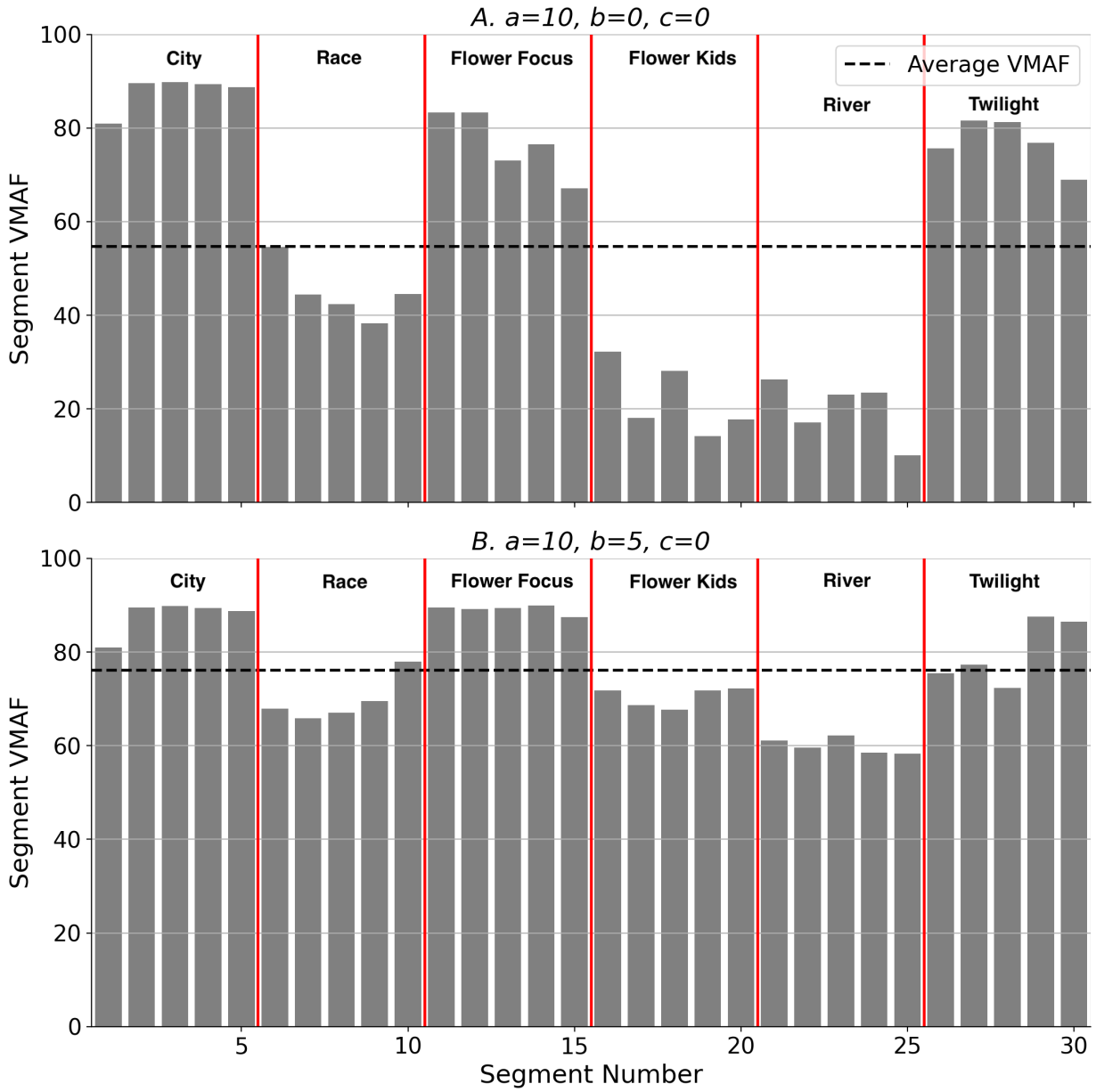


Figure 7.3: Segment wise VMAF distribution for a=10, b=0 and c=0 and for a=10, b=5 and c=0 when the bandwidth was set to 5000 kbps

the buffer to a healthy state again by transcoding and transmitting easy to encode `Twilight` sequences in a time less than the segment duration resulting in a buffer recovery.

7.2.2 Impact of b

Fig. 7.6 and Fig. 7.7 show the average stream bitrate and stream VMAF for different bandwidth profiles when a was set to 10, c was set to 0, and b was varied from 0 to 10. As the b value increases from 0 to 10, antifreeze starts delivering higher quality segments that need more encoding bits. Consequently, the stream bitrate and stream VMAF increases with increasing b. For the 5000 kbps stream, the VMAF value increased from 54.69 to 78.52 with b increasing from 0 to 10 indicating that stream quality approaches the target quality (*i.e.,* 90) with increasing b.

However, the increase in VMAF with increasing b comes at the cost of `Antifreeze` selecting high quality variants encoded with more bits which require more transcoding time and more download time. Fig. 7.8 shows the average response time (*i.e.,* transcoding time + download time) and wait time for 5000 kbps stream when a was set to 10, c was set to 0 and b was varied from 0 to 10. As b increases, both the transcoding time and the download time increases. The transcoding time increases because more processing (transcoding time) is needed to encode videos with higher quality. Download time increases because the segment bitrate increases. With the transcoding time and download time increasing, wait times also increases as the rate at which segments are being delivered to the client is lower than the live video rate. This causes live segments to be queued by at the server waiting for the client's request resulting in wait times.

The wait time, transcoding time, and download time also depend on the segment complexity. Fig. 7.9 shows the response time (transcoding time + download time) and wait times for different video sequences with $a = 10$, $b = 6$, and $c = 0$. We observe that `Race` `FlowerKids`, and `River`, segments have a response time (*i.e.,* transcoding time plus download time) that is always higher than 2 seconds (*i.e.,* segment duration). This leads to

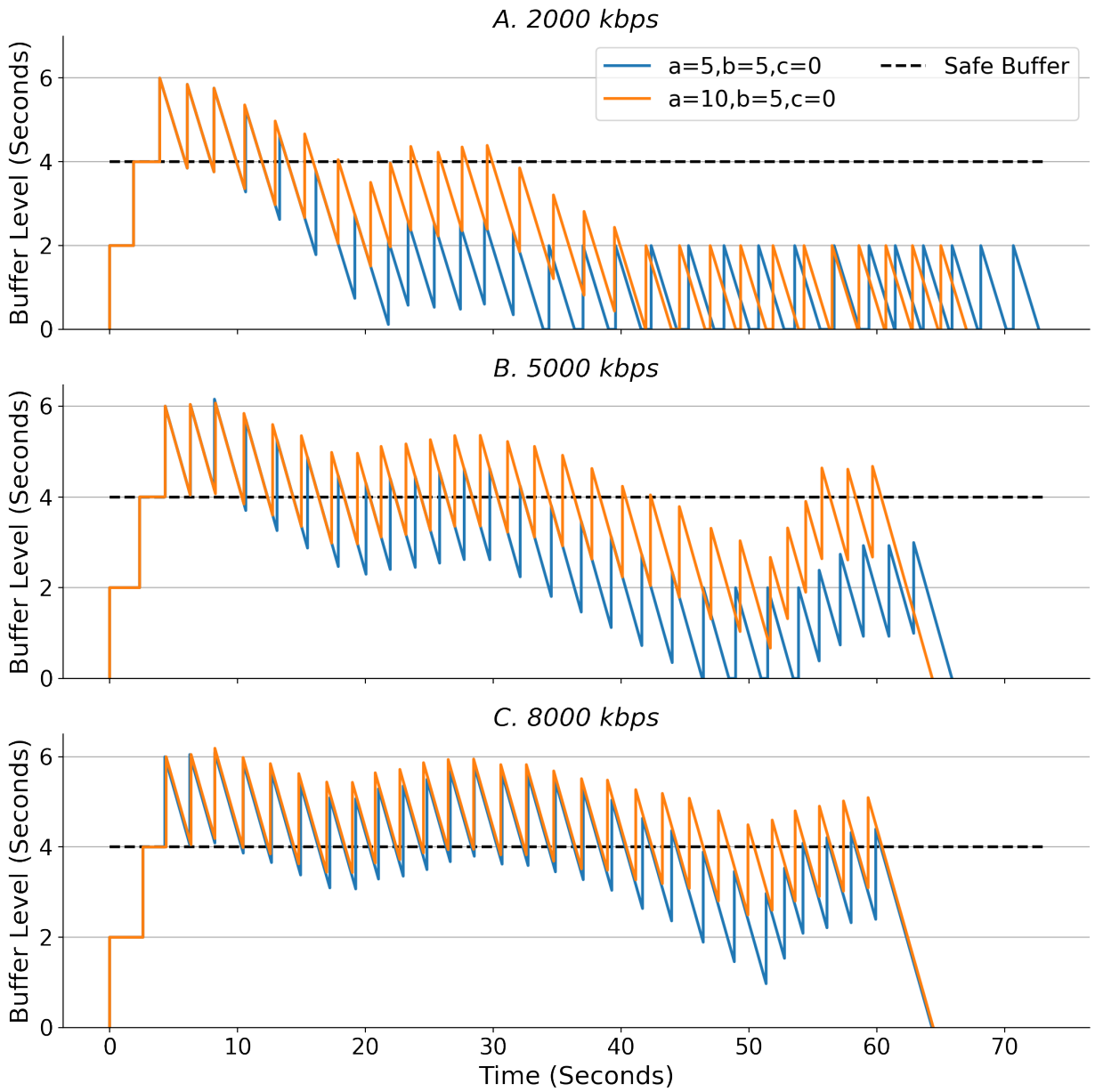


Figure 7.4: Buffer behavior for a=5, b=5 and c=0 and for a=10, b=5 and c=0 under different bandwidth profiles

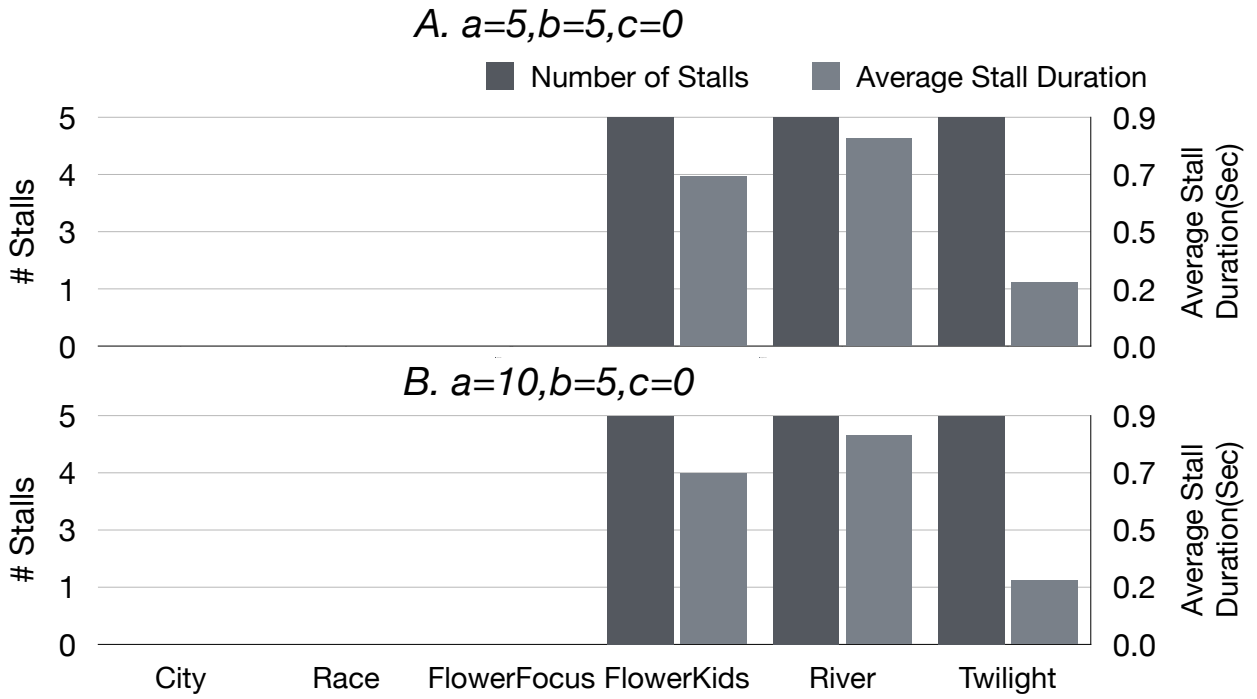


Figure 7.5: Sequence wise stall distribution for a=5, b=5 and c=0 and a=5, b=5 and c=0 when bandwidth was set to 2000 kbps

increased wait times for subsequent sequences. For the `Twilight` sequence, the response time is lower than 2 seconds implying that `Antifreeze` is recovering the buffer by delivering segments at a rate faster than the playback rate resulting in a buffer recovery.

7.2.3 Impact of c

Coefficient c minimizes the quality difference between consecutive streamed segments resulting in an overall stabilization of the streaming quality. Fig. 7.10 depicts the quality of streamed segments with $a = 10$, $b = 2$, $c = 0$ and $a = 10$, $b = 2$, and $c = 10$ when the streaming bandwidth was set to 5000 kbps. When c was set to 0, there were sudden quality changes especially, at the video sequences boundaries. But with c set to 10, the quality changes became gradual. As c is increased from 0 to 10, the standard deviation of the difference between VMAF values of consecutive segments dropped from 8.05 to 3.57. However, because c restricts sudden quality changes, it leads to a lower overall VMAF score since it

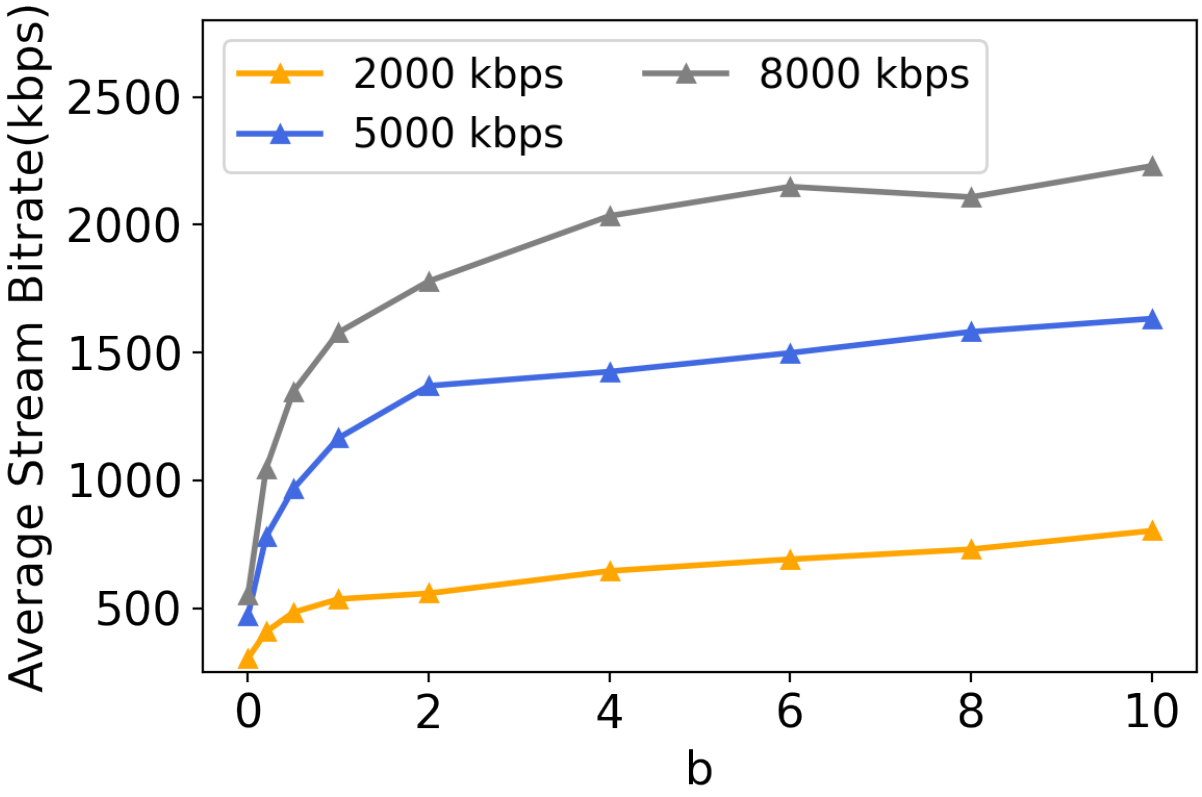


Figure 7.6: Impact of b on the stream bitrate when $a = 10$, $c = 0$ under different bandwidth profiles

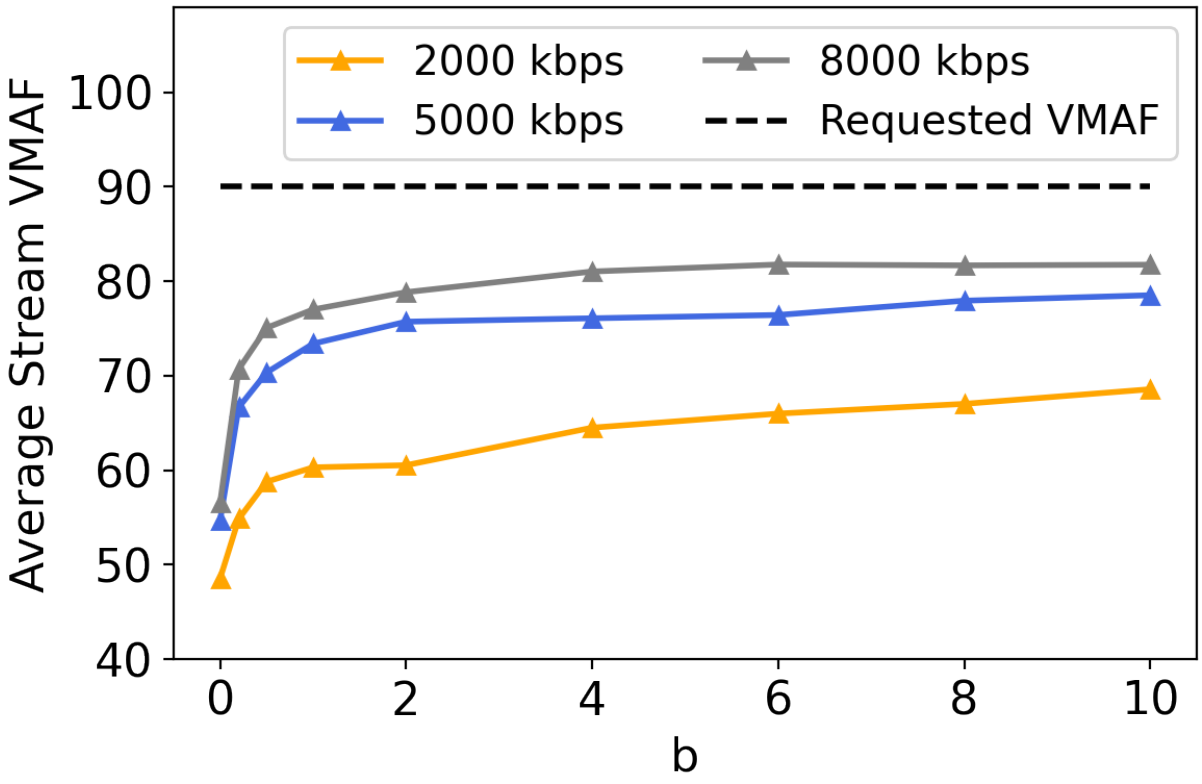


Figure 7.7: Impact of b on the stream quality VMAF when $a = 10$, $c = 0$ under different bandwidth profiles

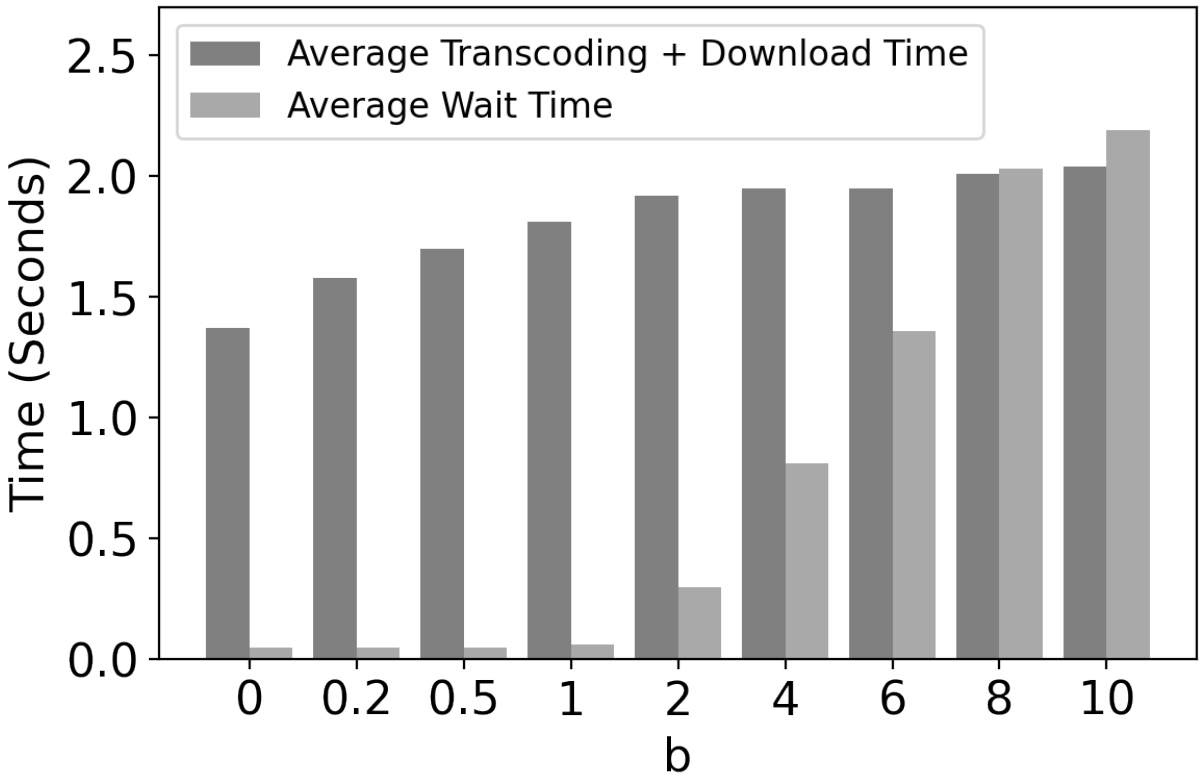


Figure 7.8: Impact of b on the wait time and response time (transcoding time + download time) for $a = 10$, $c = 0$ under a bandwidth of 5000 kbps

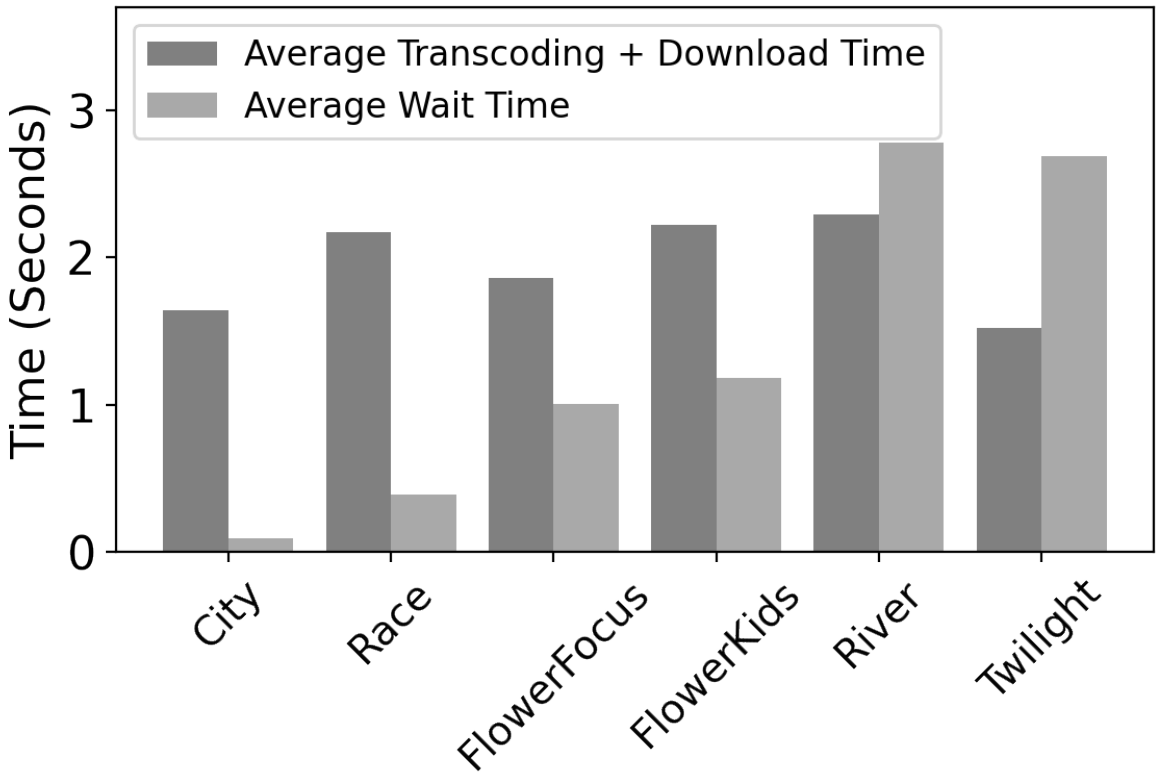


Figure 7.9: Sequence wise distribution of Wait time and response time (transcoding time + download time) for $a = 10$, $b = 6$, $c = 0$ under a bandwidth of 5000 kbps

does not permit a sudden rise in quality even if the transcoding and bandwidth allow for that. The average VMAF for $b = 0$ was 75.72, The average drops to 70.3 when b was updated to 10.

7.3 Comparison of Antifreeze with Baseline Algorithms

We compare Antifreeze with two baseline algorithms. The first baseline algorithm denoted as *baseline*1 is optimized to always keep the client buffer above the safe buffer level to ensure playback smoothness. An effective strategy to achieve this goal is to make the rate at which the buffer is being emptied out due to playback to be less than or equal to the rate at which new content is being fetched from the server. For segments d seconds long, this implies that each segment has to be transcoded and delivered to the client in a time less than or equal to d seconds. To maximize the delivered quality, the server can transcode a segment to a variant whose quality is closest to the user-requested quality but can still be transcoded and delivered to the client within d seconds.

The second baseline algorithm denoted as *baseline*2 is optimized for maximizing the video quality and delivers those variants to the clients whose quality level is closest to the target quality q_i irrespective of the client's buffer level and available bandwidth. This approach maximizes the delivered video quality but is susceptible to stalls especially when the target quality level is high.

Fig. 7.11 shows the buffer behavior for *baseline*1, *baseline*2, and `Antifreeze` ($a = 10, b = 2, c = 4$) when the bandwidth was set to 5000 kbps. With *baseline*1, the buffer always stays above the safe level, whereas the buffer in *baseline*2 is never in a healthy state which results in delays and playback stalls. *baseline*2 always streams close to the target quality q_i, whereas `Antifreeze` adapts to the segment complexity and adjusts the transcoding strategy in real-time to generate good quality segments without risking playback stalls.

Table 7.2 Compares `Antifreeze` (with a=10, b=2,c=4) with the two baseline algorithms.

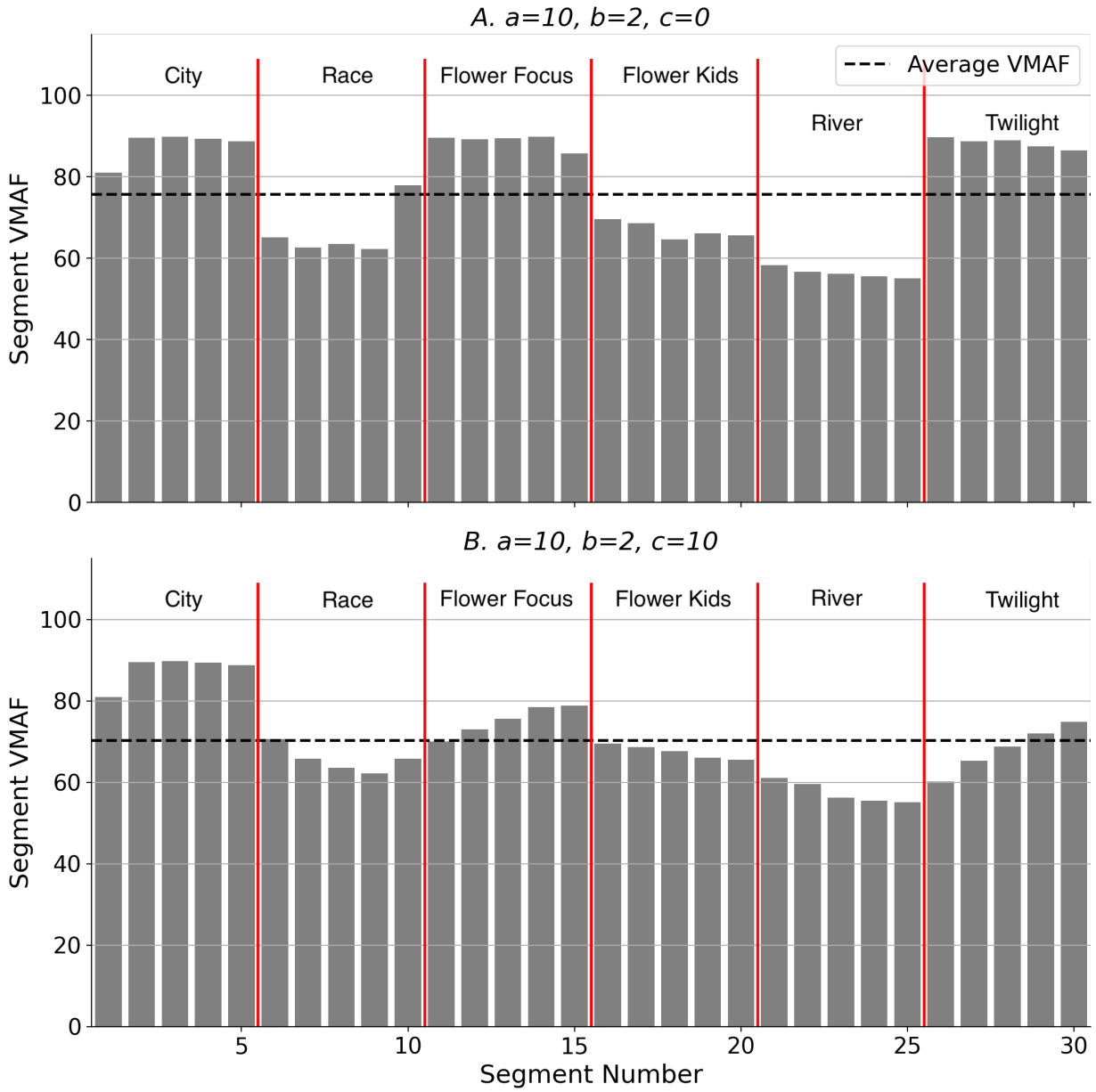


Figure 7.10: Segment wise VMAF distribution for a=10, b=2 and c=0 and for a=10, b=2 and c=10 when the bandwidth was set to 5000 kbps

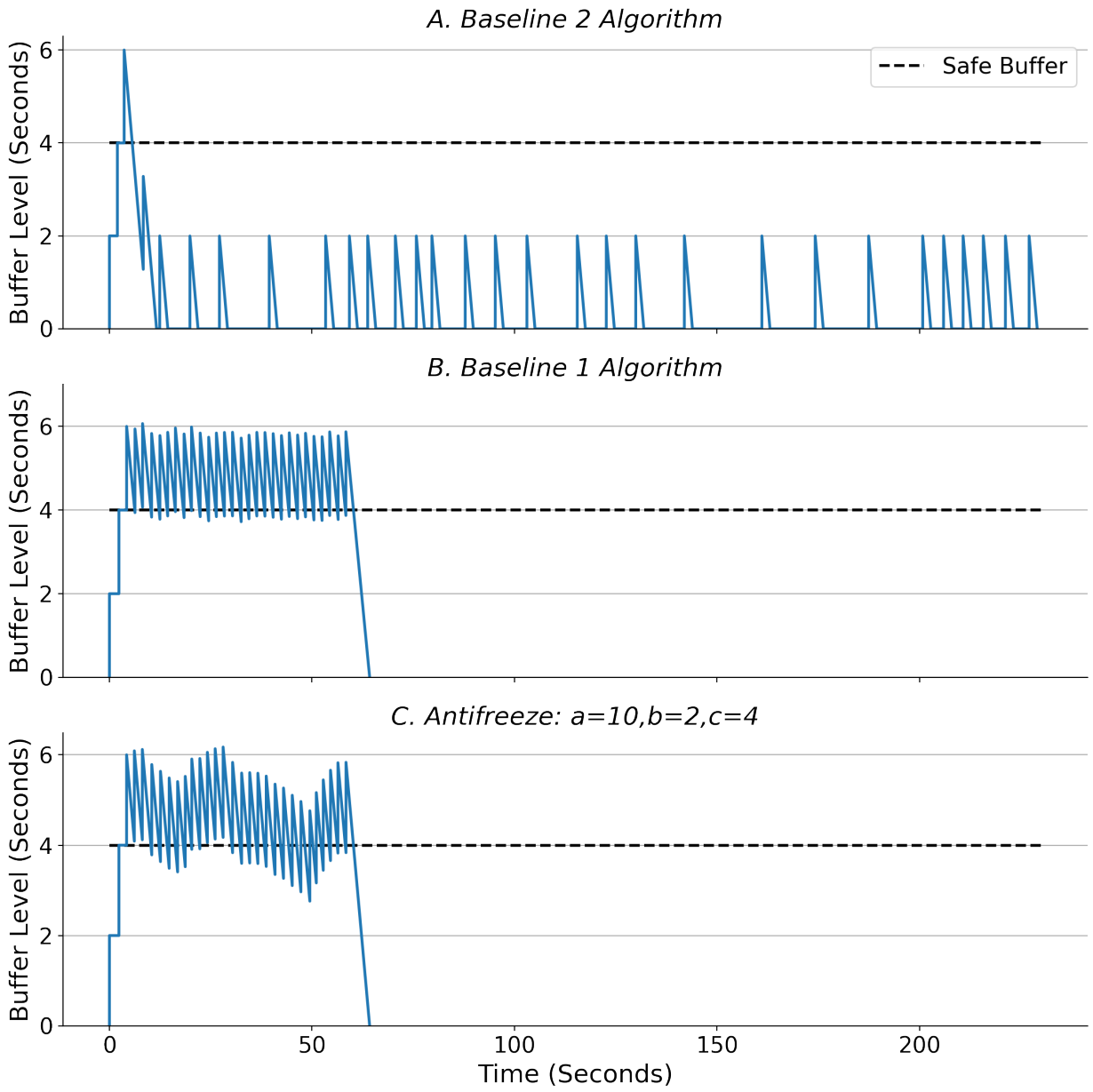


Figure 7.11: Buffer level when comparing `Antifreeze` with Baseline Algorithm 1, Baseline Algorithm 2. The bandwidth was set to 5000 kbps.

Table 7.2: Comparison of Antifreeze (with a=10, b=2,c=4) with baseline algorithms b_1 when the bandwidth was set to 5000 kbps

S.N	Algorithm	Stream VMAF	# Stalls	Average Stall Duration (Seconds)
1	Baseline 1	73.57	0	0
2	Baseline 2	87.38	26	6.36
3	Antifreeze	73.12	0	0

*baseline*1 and `Antifreeze` have identical average VMAF score, and *baseline*2 has a significantly higher VMAF scores. `Antifreeze` however outperforms *baseline*1 when it comes to consistency in video quality. This is because `Antifreeze` allows transcoding time flexibility, resulting in an overall smooth stream. *baseline*2 is impractical since it has an unacceptable number of stalls.

7.4 Variable Bandwidth Streaming

At last, we evaluate the behavior of `Antifreeze` under variable bandwidth profiles. Specifically, we investigate the buffer behavior when the bandwidth changes. We selected following three bandwidth profiles:

1. bandwidth-drop profile: the bandwidth drops from 5000 kbps to 2000 kbps and then stays at 2000 kbps.

2. bandwidth-rise profile: the bandwidth rises from 5000 kbps to 8000 kbps and then stays at 8000 kbps.

3. cyclic bandwidth profile: the bandwidth cycles between 5000 kbps and 1000 kbps.

We know from our evaluation of coefficients a, b, and c, that the optimal coefficient values depend on the available bandwidth. Therefore for our variable bandwidth performance analysis, we dynamically adapt the coefficients as well. Table 7.3 shows the chosen coefficient values for different bandwidths. The value of b is high at higher bandwidth because at high

Table 7.3: coefficient a, b and c at different bandwidth

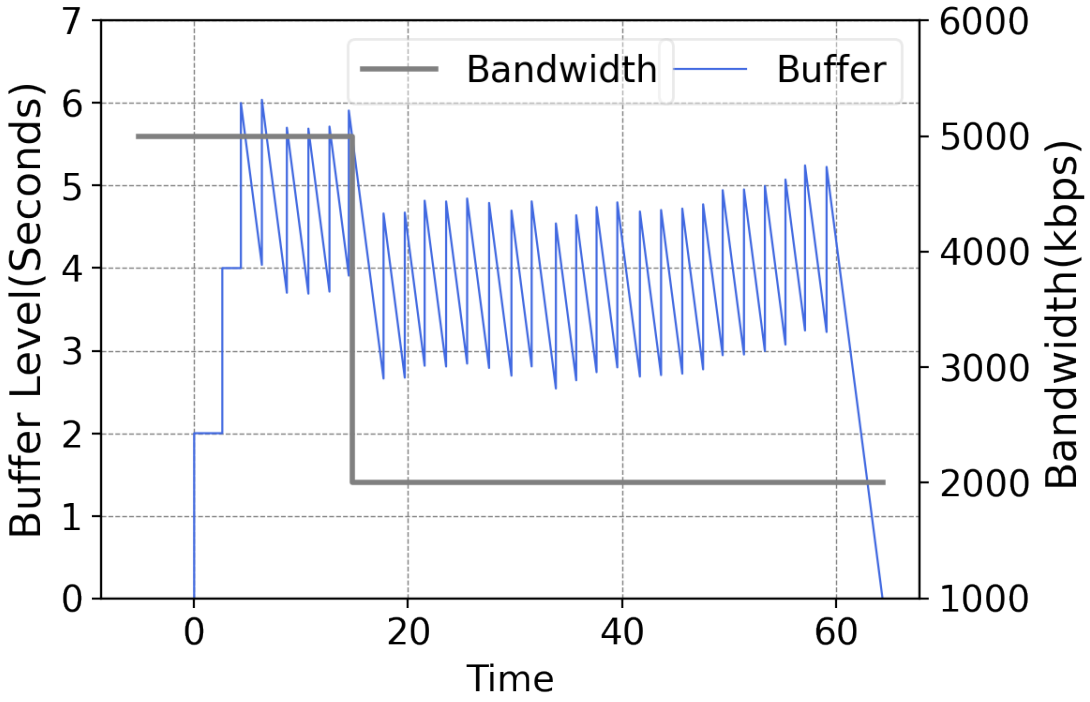
S.N	Bandwidth (kbps)	a	b	c
1	>= 8000	10	4	2
2	>= 6000	10	3	2
3	>=4000	10	1	1
4	>=2000	10	1	0.5
5	Else	15	0.5	0.5

bandwidths high bitrate segments can be streamed without risking playback stalls. For low bandwidth, the adaptation is conservative with a high a and a low b value to keep the buffer level healthy. Therefore, based on the observed bandwidth, coefficients a, b, and c may change over time during the streaming session.

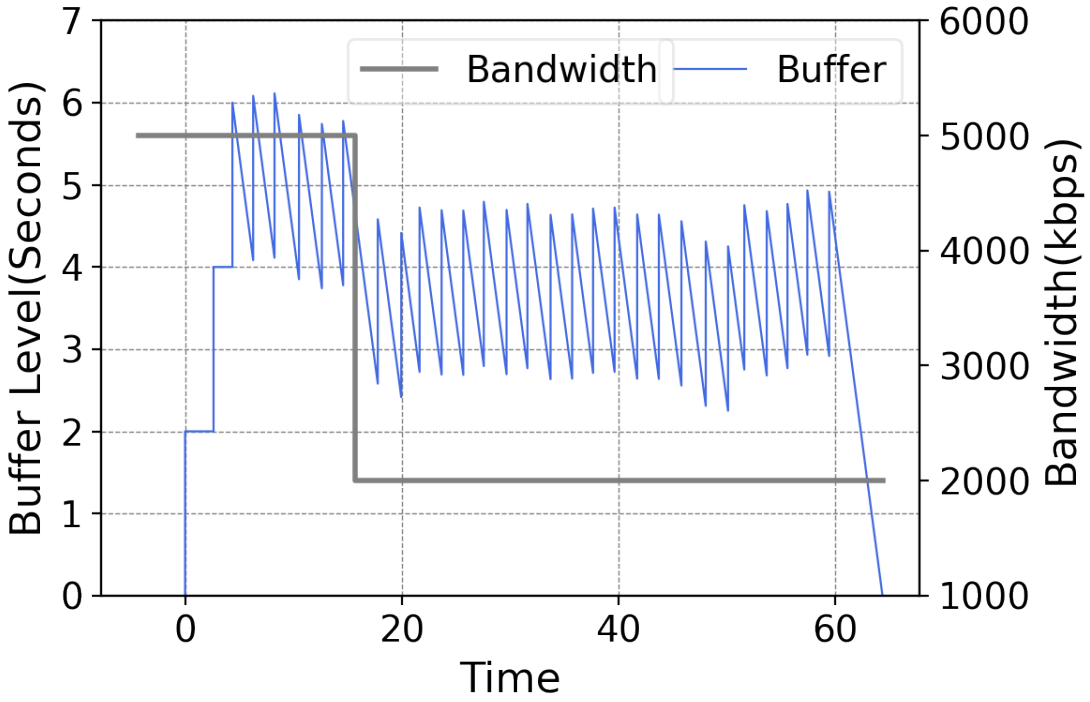
Fig. 7.12, Fig. 7.13 and, Fig. 7.14 compare the buffer behavior for *baseline*1 and `Antifreeze` for bandwidth-drop profile, bandwidth-rise profile, and cyclic bandwidth profile, respectively. We observe that the buffer level drops under the bandwidth-drop profile for both *baseline*1 and `Antifreeze`, as shown in Fig. 7.12. For the bandwidth-rise profile, the buffer stayed above 4 seconds with *baseline*1 but dropped with `Antifreeze` Fig. 7.13. This drop in `Antifreeze` is not a worrying sign as `Antifreeze` can bring the buffer to safe levels again. For the cyclic bandwidth profile *baseline*1 resulted in lots of stalls (more precisely, 11 stalls) whereas `Antifreeze` has much fewer stalls (only 2). `Antifreeze` has fewer stalls because it can recover the buffer after stalls whereas *baseline*1 cannot recover the buffer once it dropped.

7.5 Practicality of Antifreeze

In this chapter, we evaluated the QoE benefits offered by `Antifreeze` by means of run-time adaptation. To determine the most optimal transcoded variant (*i.e.,* CRF-resolution pair), we fed a pre-computed lookup table (which contains the bitrate and VMAF score of all possible transcoded variants) to the S-Controller module. Then to simulate the transcoding process, we introduced a `wait()` equalling the transcoding time of the selected transcoded

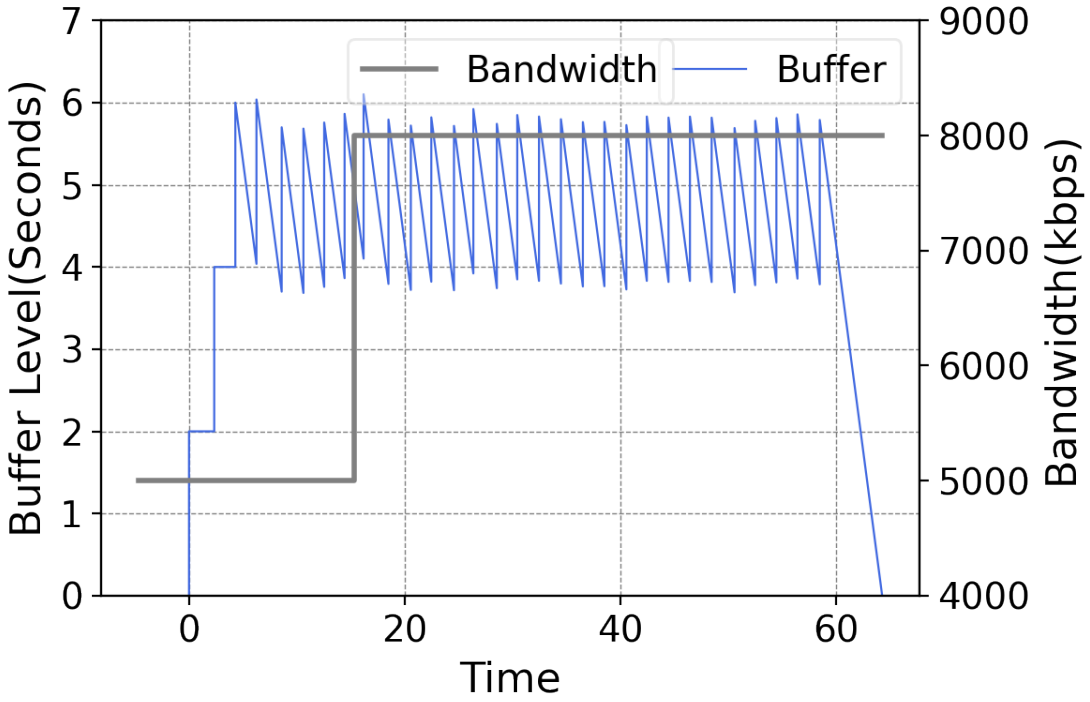


(a) Baseline 1 Algorithm

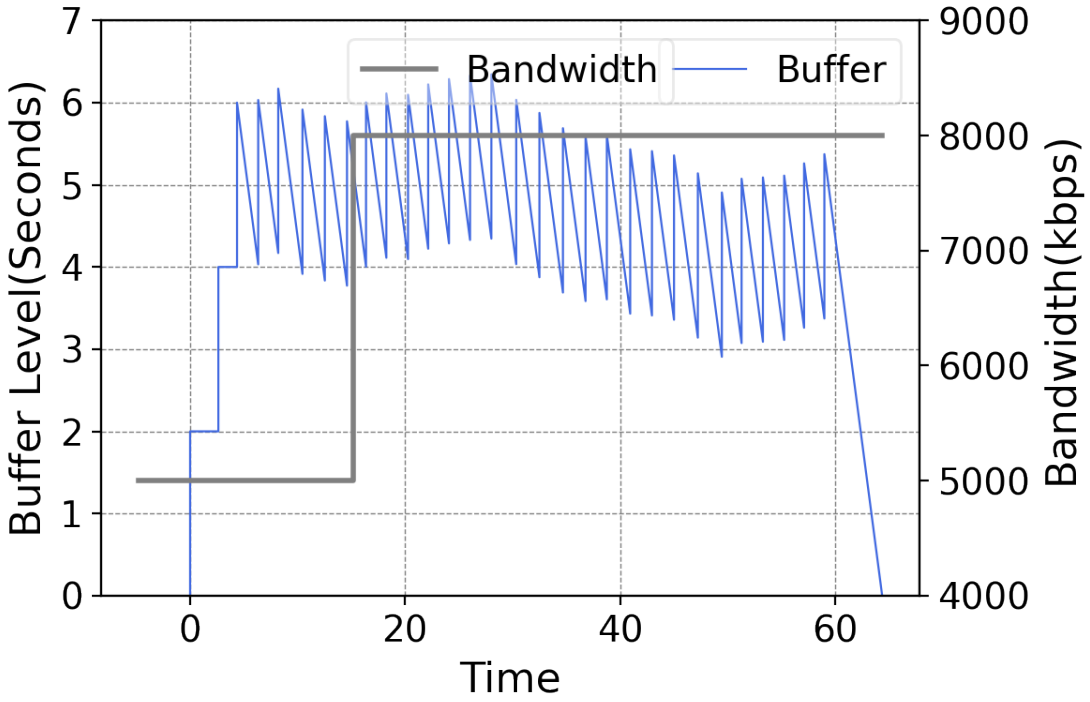


(b) Antifreeze

Figure 7.12: Buffer behavior comparison for Baseline 1 Algorithm and `Antifreeze` under bandwidth-drop profile

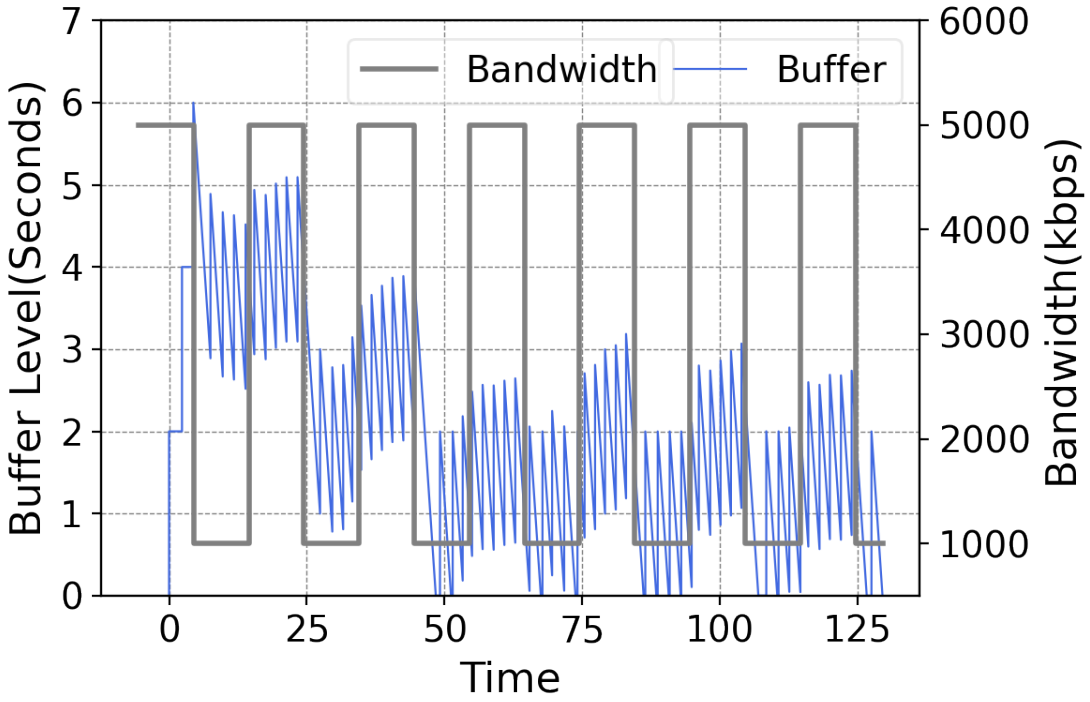


(a) Baseline 1 Algorithm

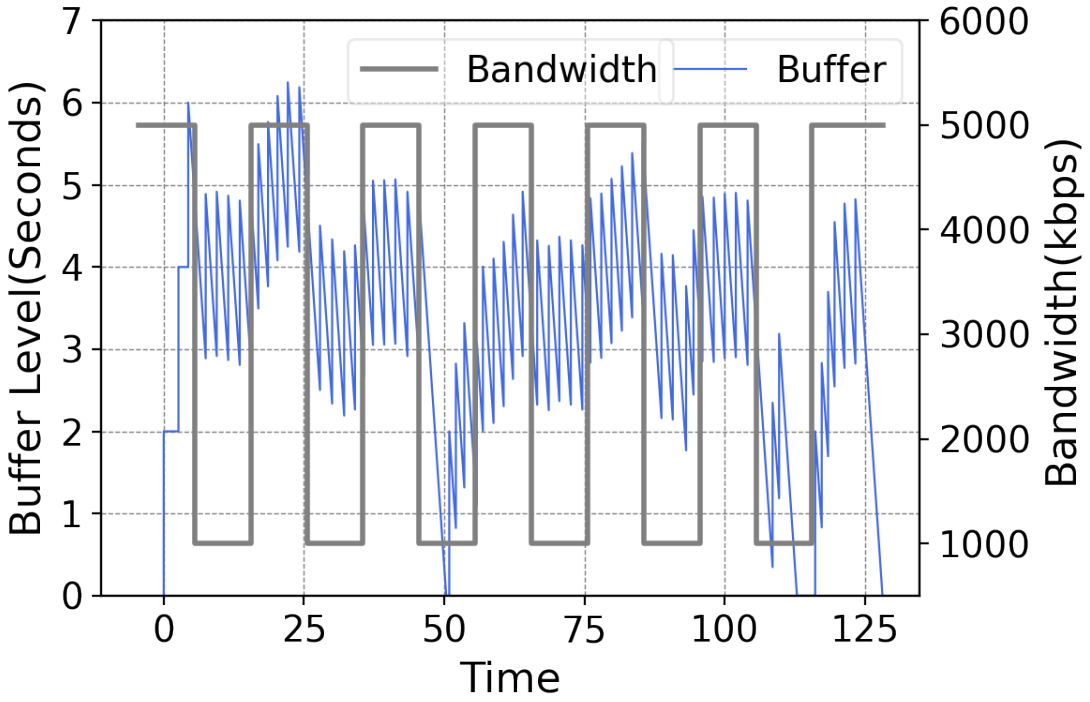


(b) Antifreeze

Figure 7.13: Buffer behavior comparison for Baseline 1 Algorithm and `Antifreeze` under bandwidth-rise profile



(a) Baseline 1 Algorithm



(b) Antifreeze

Figure 7.14: Buffer behavior comparison for Baseline 1 Algorithm and `Antifreeze` under Cyclic bandwidth profile profile

variant (CRF-resolution pair). The lookup table mode of evaluation gives a true performance evaluation of antifreeze as there are no inaccuracies in the adaptation algorithm as inputs to the algorithm (S-Controller) comes from actual transcoding stats. However, for real world deployments, the lookup table should be replaced by Machine Learning models. Specifically, three models are needed - transcoding time prediction model, bitrate prediction model and a visual quality (VMAF) prediction model. Integratying prediction models is out of scope of this thesis and is something to be addressed in future work.

The current design of Antifreeze also suffers from scalability issues as the S-Controller makes segment quality decisions on a per client basis. Strategies to make the system more scalable needs to be devised in the future. Some possible strategies include techniques to batch similar kind of user requests and horizontal scaling (increasing the number of transcoder) to cater to the needs of multiple clients.

7.6 Summary

Coefficients a, b, and c are the weights assigned to the buffer loss (ΔB), quality loss (ΔQ), and quality switch loss(ΔS), respectively. Specifically, a tries to keep the buffer in a healthy state to ensure that the playback does not suffer from stalls; b tries to match the stream quality to the user-requested quality to ensure that the user gets what they requested, and c tries to keep the stream quality consistent by stabilizing the stream quality and not allowing sudden quality switches. However, a, b and c present conflicting challenges, and increasing one of them leads to a decrease in the impact of the other(s). For instance, if b is increased, then the algorithm will deliver higher quality segments. However, generating high-quality encodes requires more transcoding time and more encoding bits. This can potentially result in an overall decrease in the delivery rate of segments, which can lead to stalls – defeating the objective of a. Therefore establishing a good balance between the coefficients is crucial. If the coefficients are balanced, each of the coefficients plays at its advantage to optimize the

overall QoE. For instance, for $a = 10$, $b = 6$, and $c = 0$ under a bandwidth of 5000 kbps as shown in Fig. 7.4(B), we observe that the buffer drops when encoding complex segments in high quality (impact of b). However, as soon as simple segments were streamed, a started recovering the buffer towards safe level.

The available bandwidth also plays a crucial factor in determining optimal a, b, and c values. Essentially, high bandwidths open up the possibility to stream high-quality segments without risking stalls. Thus b becomes more significant at higher bandwidths. The optimal a, b, and c values also differ from segment to segment since segments have different transcoding needs in terms of complexity. The more complex a segment (*e.g.,* segments in `Race`, `FlowerKids`, `River`) is, the more challenging the real-time transcoding is and the more difficult in keeping the buffer level healthy.

Finally, Our results show that Antifreeze significantly reduces the playback stalls and substantially improves the visual quality in interactive video streaming sessions under various bandwidth profiles and outperforms other baseline algorithms.

Chapter 8

Conclusion

This chapter gives an overview of the research conducted in this thesis and discusses future research goals. Sec. 8.1 provides a summary of the research and Sec. 8.2 discusses future works.

8.1 Thesis Summary and Conclusion

This research focused on improving the QoE of a live stream given the transcoding challenges. Specifically, the goal was to design and implement a real-time transcoding and streaming solution which can dynamically adapt to the client's needs during run-time to provide a more streamlined viewing experience to the viewer.

Transcoding is a time-consuming and resource-intensive process. It is often the transcoding process that decides if a video can be streamed in real-time. To minimize the transcoding delay we evaluated the tools offered by the codec (*i.e.,* `libx265`) to speed up the transcoding process. Our results suggested that transcoding a video segment on multiple threads using modes such as inter-frame parallelism and wavefront parallel processing (WPP) can lead to significant transcoding speedups. Additionally, we observed that depending on the transcoding complexity, the transcoder throughput may not be sufficient to sustain a live stream and there is a need for distributed transcoding in such cases. Consequently, to ease distributed

transcoding we implemented docker-based transcoding as docker containers can be deployed easily on the cloud and can leverage a large amount of computational resources in a scalable manner. Additionally, with docker, it is possible to limit the amount of resources a container can use resulting in resource efficiency.

To facilitate real-time transcoding on docker containers while keeping the process resource efficient we proposed Machine-Learning based solutions to characterize the transcoder performance. In particular, we developed two prediction models. The first model is a regression model and predicts the transcoding time of a segment based on the segment properties, the transcoder parameters, and the container size (*i.e.,* the number of CPU cores allocated to the container). The second model is a classification model and predicts the minimum container size needed to reach a certain transcoding throughput. The transcoding time prediction model had a Mean Absolute Error of 0.116 seconds and a Mean absolute percentage error of 2.191%. The CPU cores prediction model had an accuracy of 98.3% and an f-1 score of 0.97 in predicting the CPU cores required to reach a target transcoding throughput. Additionally, the models have significantly fewer overheads, and thus integrating the models in the transcoding pipeline does not introduce any significant delays. When used together, the two models can help balance the trade-off between transcoding rate and CPU cores in the context of real-time transcoding. Our results indicate that using the prediction model to provision resources leads to resource savings compared to a fixed resource allocation strategy that does not consider the per-segment transcoding performance when allocating resources.

In the second half of this thesis, we designed and implemented `Antifreeze`, an on-demand real-time transcoding and streaming platform to offer a more personalized streaming experience to the viewer. `Antifreeze` can transcode a live stream in real-time to the most optimal quality variant depending on the client's needs and the playback conditions. To the best of our knowledge, antifreeze is the first streaming platform that considers delay due to on-the-fly transcoding when determining the most optimal variant to which a requested segment should be transcoded in real-time.

To come up with the design of `Antifreeze`, we first investigated the client buffer model for a live stream transcoded on-the-fly in real-time. From the buffer analysis, we concluded that the buffer change from receiving one segment to the next one depends on the transcoding time (tt), download time (dt), and (wt) and thus determining the optimal transcoded variant is a must to ensure that the buffer does not drop to critical levels. Based on the buffer model we proposed a QoE loss function for each transcoded variant. Specifically, the loss function for a transcoded variant relates to how close the transcoded variant is to an ideal stream. We then implemented a real-time transcoding platform `Antifreeze` which consists of a HAS server and a HAS client. Upon receiving a request for a segment from the client at a certain quality level(q_i), the server transcodes the segment on the fly to the best variant which can be generated and delivered to the client in time without causing playback stalls. Specifically for a transcoding task $s \rightarrow s'$ where s is the source segment and s' is the transcoded segment the `Antifreeze` determines that s' whose quality is close to the q_i and which can be generated and delivered to the client fast to avoid playback stalls.

To determine the most optimal transcoded variant `Antifreeze` adaptation computes the buffer loss (ΔB), quality loss (ΔQ), and quality switch loss (ΔQ) for each possible transcoded variant. The Coefficients a, b, and c are the weights assigned to (ΔB), (ΔQ), and (ΔS), respectively. Specifically, a tries to keep the buffer in a healthy state to ensure that the playback does not suffer from stalls, b tries to match the stream quality to the user requested quality q_i to ensure that the user gets what he requested for and c tries to keep the stream quality consistent by stabilizing the stream quality and not allowing sudden quality switches. From our results, we observed that with the proper selection of a, b, and c, antifreeze can deliver a high-quality smooth stream to the viewer. Additionally, we also concluded that optimal a, b, and c values vary for different bandwidth profiles and different kinds of video segments.

Our evaluation results also show that `Antifreeze` outperforms other baseline algorithms in terms of delivered QoE. When the buffer level drops, either due to bandwidth drops or

due to transcoding overheads, antifreeze can bring the buffer level to safe levels again thus minimizing the chances of playback stalls.

8.2 Future Work

Following are some of the limitations of our work and possible future research directions:

1. For evaluating the performance of `Antifreeze` we used a lookup table to compute the loss function. Integrating prediction models within `Antifreeze` to make it smarter is something to be researched in the future

2. Integrating `Antifreeze` with existing HAS protocols like DASH is also something to be explored in the future.

3. A container orchestration tool like Kubernetes [125] and docker swarm [126] is suitable for running and managing multiple containers (scalable) across multiple nodes and to make the system fault tolerant. As a future work Kubernetes can be used to provide better horizontal and vertical scaling for the services and to make the system more robust. However, for this thesis, our focus was on characterizing the performance of containerized transcoder especially evaluating the relation between number of processing cores and transcoding speed. Therefore, to keep the overheads to a minimum we used bare containers to do the modelling against using an orchestration tool like Kubernetes.

4. Transcoding in not only heavy on the CPU but also heavy on RAM and Disk. Studying the relation between the transcoder performance and the RAM and Disk usage is something to be studied in the future.

5. Since 360-degree videos are computationally more demanding than 2-D videos, extending the design of `Antifreeze` to 360-degree videos is also a future research direction.

Bibliography

[1] "Global_device_growth_traffic_profiles." [Online]. Available: https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_Device_Growth_Traffic_Profiles.pdf

[2] Y. Liu, "Exploring Temporal Loss Tolerance of Video Codecs for QoE Enhancement in Adaptive Streaming."

[3] "Hevc: An introduction to high efficiency coding - vcodex." [Online]. Available: https://www.vcodex.com/hevc-an-introduction-to-high-efficiency-coding/

[4] "Hevc – what are ctu, cu, ctb, cb, pb, and tb? — code: Sequoia." [Online]. Available: https://codesequoia.wordpress.com/2012/10/28/hevc-ctu-cu-ctb-cb-pb-and-tb/

[5] I.-K. Kim, J. Min, T. Lee, W.-J. Han, and J. Park, "Block Partitioning Structure in the HEVC Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1697–1706, Dec. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6324412/

[6] C. Yan, Y. Zhang, J. Xu, F. Dai, J. Zhang, Q. Dai, and F. Wu, "Efficient Parallel Framework for HEVC Motion Estimation on Many-Core Processors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 12, pp. 2077–2089, Dec. 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6848755/

[7] S. Radicke, J.-U. Hahn, C. Grecos, and Q. Wang, "A Multi-Threaded Full-feature HEVC Encoder Based on Wavefront Parallel Processing:," in *Proceedings of the 11th International Conference on Signal Processing and Multimedia Applications*. Vienna, Austria: SCITEPRESS - Science and and Technology Publications, 2014, pp. 90–98. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005018300900098

[8] Shaobo Zhang, Xiaoyun Zhang, and Zhiyong Gao, "Implementation and improvement of Wavefront Parallel Processing for HEVC encoding on many-core platform," in *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*. Chengdu, China: IEEE, Jul. 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/6890596/

[9] P. Habermann, B. Juurlink, C. C. Chi, and M. Alvarez-Mesa, "Efficient Wavefront Parallel Processing for HEVC CABAC Decoding," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Västerås, Sweden: IEEE, Mar. 2020, pp. 339–343. [Online]. Available: https://ieeexplore.ieee.org/document/9092425/

[10] Y. Zhao and J. Wen, "Wavefront Parallel Processing for AV1 Encoder," in *2018 Picture Coding Symposium (PCS)*. San Francisco, CA: IEEE, Jun. 2018, pp. 101–105. [Online]. Available: https://ieeexplore.ieee.org/document/8456283/

[11] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, "Parallel Scalability and Efficiency of HEVC Parallelization Approaches," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1827–1838, Dec. 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6327343/

[12] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on Circuits and*

Systems for Video Technology, vol. 22, no. 12, Dec. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6316136/

[13] "Per-Title Encode Optimization." [Online]. Available: https://netflixtechblog.com/per-title-encode-optimization-7e99442b62a2

[14] U. Cisco, "Cisco annual internet report (2018–2023) white paper," *Cisco: San Jose, CA, USA*, 2020.

[15] "Nielsen's Law of Internet Bandwidth." [Online]. Available: https://www.nngroup.com/articles/law-of-bandwidth/

[16] Y. Qin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, B. Wang, and C. Yue, "Quality-aware strategies for optimizing ABR video streaming QoE and reducing data usage," in *Proceedings of the 10th ACM Multimedia Systems Conference*. Amherst Massachusetts: ACM, Jun. 2019, pp. 189–200. [Online]. Available: https://dl.acm.org/doi/10.1145/3304109.3306231

[17] Y. Qin, R. Jin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, C. Yue, and B. Wang, "A Control Theoretic Approach to ABR Video Streaming: A Fresh Look at PID-Based Rate Adaptation," *IEEE Transactions on Mobile Computing*, vol. 19, no. 11, pp. 2505–2519, Nov. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8765424/

[18] P. K. Yadav, A. Bentaleb, M. Lim, J. Huang, W. T. Ooi, and R. Zimmermann, "Playing chunk-transferred DASH segments at low latency with QLive," in *Proceedings of the 12th ACM Multimedia Systems Conference*. Istanbul Turkey: ACM, Jul. 2021, pp. 51–64. [Online]. Available: https://dl.acm.org/doi/10.1145/3458305.3463376

[19] "Live video streaming: How it works — wowza." [Online]. Available: https://www.wowza.com/blog/live-video-streaming-how-it-works

[20] "What is a video cdn? — cdn video streaming — cloudflare." [Online]. Available: https://www.cloudflare.com/en-gb/learning/video/what-is-video-cdn/

[21] "What is video transcoding and how to use it — adobe." [Online]. Available: https://www.adobe.com/mt/creativecloud/video/discover/video-transcoding.html

[22] "Video transcoding - what is transcoding?" [Online]. Available: https://www.gumlet.com/learn/video-transcoding-what-is-transcoding/

[23] A. Zabrovskiy, P. Agrawal, R. Matha, C. Timmerer, and R. Prodan, "ComplexCTTP: Complexity Class Based Transcoding Time Prediction for Video Sequences Using Artificial Neural Network," in *2020 IEEE Sixth International Conference on Multimedia Big Data (BigMM)*. New Delhi, India: IEEE, Sep. 2020, pp. 316–325. [Online]. Available: https://ieeexplore.ieee.org/document/9232616/

[24] "What is adaptive bitrate streaming? — cloudflare." [Online]. Available: https://www.cloudflare.com/en-gb/learning/video/what-is-adaptive-bitrate-streaming/

[25] H. Zhao, Q. Zheng, W. Zhang, B. Du, and H. Li, "A Segment-Based Storage and Transcoding Trade-off Strategy for Multi-version VoD Systems in the Cloud," *IEEE Transactions on Multimedia*, vol. 19, no. 1, pp. 149–159, Jan. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7572977/

[26] M. Darwich, E. Beyazit, M. A. Salehi, and M. Bayoumi, "Cost Efficient Repository Management for Cloud-Based On-demand Video Streaming," in *2017 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. San Francisco, CA, USA: IEEE, Apr. 2017, pp. 39–44. [Online]. Available: http://ieeexplore.ieee.org/document/7944870/

[27] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang, "QDASH: a QoE-aware DASH system," p. 12.

[28] Y. Qin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, B. Wang, and C. Yue, "ABR streaming of VBR-encoded videos: characterization, challenges, and solutions," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. Heraklion Greece: ACM, Dec. 2018, pp. 366–378. [Online]. Available: https://dl.acm.org/doi/10.1145/3281411.3281439

[29] Z. Li, A. C. Begen, J. Gahm, Y. Shan, B. Osler, and D. Oran, "Streaming video over HTTP with consistent quality," in *Proceedings of the 5th ACM Multimedia Systems Conference on - MMSys '14*. Singapore, Singapore: ACM Press, 2014, pp. 248–258. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2557642.2557658

[30] F. Zheng, Z. Shi, X. Zhang, and Z. Gao, "Fast H.264/AVC to HEVC transcoding based on residual homogeneity," in *2014 International Conference on Audio, Language and Image Processing*. Shanghai, China: IEEE, Jul. 2014, pp. 765–770. [Online]. Available: http://ieeexplore.ieee.org/document/7009898/

[31] A. J. Diaz-Honrubia, J. L. Martinez, J. M. Puerta, J. A. Gamez, J. De Cock, and P. Cuenca, "Fast quadtree level decision algorithm for H.264/HEVC transcoder," in *2014 IEEE International Conference on Image Processing (ICIP)*. Paris: IEEE, Oct. 2014, pp. 2497–2501. [Online]. Available: https://ieeexplore.ieee.org/document/7025505/

[32] G. Correa, L. Agostini, and L. A. da Silva Cruz, "Fast H.264/AVC to HEVC transcoder based on data mining and decision trees," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. Montréal, QC, Canada: IEEE, May 2016, pp. 2539–2542. [Online]. Available: https://ieeexplore.ieee.org/document/7539110

[33] S. Wang, S. Ma, S. Wang, D. Zhao, and W. Gao, "Fast multi reference frame motion estimation for high efficiency video coding," in *2013 IEEE International Conference*

on Image Processing. Melbourne, Australia: IEEE, Sep. 2013, pp. 2005–2009. [Online]. Available: http://ieeexplore.ieee.org/document/6738413/

[34] G. Gao and Y. Wen, "Morph: A Fast and Scalable Cloud Transcoding System," in *Proceedings of the 24th ACM international conference on Multimedia*. Amsterdam The Netherlands: ACM, Oct. 2016, pp. 1160–1163. [Online]. Available: https://dl.acm.org/doi/10.1145/2964284.2973792

[35] S. Sameti, M. Wang, and D. Krishnamurthy, "Stride: Distributed Video Transcoding in Spark," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. Orlando, FL, USA: IEEE, Nov. 2018, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8711214/

[36] C. Song, W. Shen, L. Sun, Z. Lei, and W. Xu, "Distributed video transcoding based on MapReduce," in *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*. Taiyuan, China: IEEE, Jun. 2014, pp. 309–314. [Online]. Available: http://ieeexplore.ieee.org/document/6912152/

[37] Y.-H. Kim, J. Huh, and J. Jeong, "Distributed Video Transcoding System for 8K 360° VR Tiled Streaming Service," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. Jeju: IEEE, Oct. 2018, pp. 592–595. [Online]. Available: https://ieeexplore.ieee.org/document/8539372/

[38] S. Sameti, M. Wang, and D. Krishnamurthy, "Contrast: container-based transcoding for interactive video streaming," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[39] K. Spiteri, R. Sitaraman, and D. Sparacio, "From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 15, no. 2s, pp. 1–29, Apr. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3336497

[40] "Abr logic - dash-industry-forum/dash.js wiki." [Online]. Available: https://github-wiki-see.page/m/Dash-Industry-Forum/dash.js/wiki/ABR-Logic

[41] "Runtime options with memory, cpus, and gpus — docker documentation." [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/

[42] "What is transcoding and why is it critical for streaming? — wowza." [Online]. Available: https://www.wowza.com/blog/what-is-transcoding-and-why-its-critical-for-streaming

[43] "Introduction to color spaces in video — matrox video." [Online]. Available: https://www.matrox.com/en/video/media/guides-articles/introduction-color-spaces-video

[44] "Motion compensation - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Motion_compensation

[45] "I-p-b frames and gops – mpeg-2 compression — learn media tech." [Online]. Available: https://learnmediatech.com/i-p-b-frames-and-gops-mpeg-2/

[46] "I-p-b frames and gops – mpeg-2 compression — learn media tech." [Online]. Available: https://learnmediatech.com/i-p-b-frames-and-gops-mpeg-2/

[47] "I, p, and b-frames - differences and use cases made easy - ottverse." [Online]. Available: https://ottverse.com/i-p-b-frames-idr-keyframes-differences-usecases/

[48] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, "Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC)," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1669–1684, Dec. 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6317156/

[49] D. Grois, D. Marpe, A. Mulayoff, B. Itzhaky, and O. Hadar, "Performance comparison of H.265/MPEG-HEVC, VP9, and H.264/MPEG-AVC encoders," in *2013 Picture*

Coding Symposium (PCS). San Jose, CA, USA: IEEE, Dec. 2013, pp. 394–397. [Online]. Available: https://ieeexplore.ieee.org/document/6737766/

[50] "Bitrate vs. resolution - which is more important for video streaming? - ottverse." [Online]. Available: https://ottverse.com/bitrate-vs-resolution-video-streaming-compression/#:~:text

[51] T. Installations and L. Line, "Itu-tp. 910," *Subjective video quality assessment methods for multimedia applications, Recommendation ITU-T*, p. 910.

[52] O. Vincent and O. Folorunso, "A Descriptive Algorithm for Sobel Image Edge Detection," 2009. [Online]. Available: https://www.informingscience.org/Publications/3351

[53] "Calculate psnr, vmaf, ssim using ffmpeg - ottverse." [Online]. Available: https://ottverse.com/calculate-psnr-vmaf-ssim-using-ffmpeg/

[54] "Vmaf is hackable: What now? - streaming learning center." [Online]. Available: https://streaminglearningcenter.com/blogs/vmaf-is-hackable-what-now.html

[55] N. T. Blog, "Toward A Practical Perceptual Video Quality Metric," Apr. 2017. [Online]. Available: https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652

[56] "Vmaf: The journey continues. by zhi li, christos bampis, julie... — by netflix technology blog — netflix techblog." [Online]. Available: https://netflixtechblog.com/vmaf-the-journey-continues-44b51ee9ed12

[57] D. Nandakumar, Y. Wu, H. Wei, and A. Ten-Ami, "On the accuracy of video quality measurement techniques," in *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*. Kuala Lumpur, Malaysia: IEEE, Sep. 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8901796/

[58] A. Antsiferova, A. Yakovenko, N. Safonov, D. Kulikov, A. Gushin, and D. Vatolin, "Objective video quality metrics application to video codecs comparisons: choosing the best for subjective quality estimation," Jul. 2021, arXiv:2107.10220 [cs]. [Online]. Available: http://arxiv.org/abs/2107.10220

[59] "Rtmp streaming: What is the real-time messaging protocol?" [Online]. Available: https://www.wowza.com/blog/rtmp-streaming-real-time-messaging-protocol

[60] R. Pantos and W. May, "Http live streaming," Tech. Rep., 2017.

[61] I. Sodagar, "The MPEG-DASH Standard for Multimedia Streaming Over the Internet," *IEEE Multimedia*, vol. 18, no. 4, pp. 62–67, Apr. 2011. [Online]. Available: http://ieeexplore.ieee.org/document/6077864/

[62] "Mpeg-dash: Dynamic adaptive streaming over http — wowza." [Online]. Available: https://www.wowza.com/blog/mpeg-dash-dynamic-adaptive-streaming-over-http#:~:text=Netflix

[63] "1.1.1. Brief History of Virtualization." [Online]. Available: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html

[64] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[65] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2015, pp. 171–172.

[66] "Understanding docker "container host" vs. "container os" for linux and windows containers." [Online]. Available: http://www.floydhilton.com/docker/2017/03/31/Docker-ContainerHost-vs-ContainerOS-Linux-Windows.html

[67] V. Gupta, K. Kaur, and S. Kaur, "Performance comparison between light weight virtualization using docker and heavy weight virtualization," *Mar*, vol. 2, pp. 211–216, 2017.

[68] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014.

[69] "FFmpeg." [Online]. Available: https://www.ffmpeg.org/

[70] T. Deneke, S. Lafond, and J. Lilius, "Analysis and Transcoding Time Prediction of Online Videos," in *2015 IEEE International Symposium on Multimedia (ISM)*. Miami, FL: IEEE, Dec. 2015, pp. 319–322. [Online]. Available: http://ieeexplore.ieee.org/document/7442349/

[71] T. Deneke, "Proactive management of video transcoding services," 2017.

[72] X. Li, M. A. Salehi, and M. Bayoumi, "VLSC: Video Live Streaming Using Cloud Services," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. Atlanta, GA, USA: IEEE, Oct. 2016, pp. 595–600. [Online]. Available: http://ieeexplore.ieee.org/document/7723747/

[73] X. Li, M. A. Salehi, Y. Joshi, M. K. Darwich, B. Landreneau, and M. Bayoumi, "Performance Analysis and Modeling of Video Transcoding Using Heterogeneous Cloud Services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 910–922, Apr. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8466657/

[74] D. K. Krishnappa, M. Zink, and R. K. Sitaraman, "Optimizing the video transcoding workflow in content delivery networks," in *Proceedings of the 6th ACM Multimedia*

Systems Conference. Portland Oregon: ACM, Mar. 2015, pp. 37–48. [Online]. Available: https://dl.acm.org/doi/10.1145/2713168.2713175

[75] H. Ma, B. Seo, and R. Zimmermann, "Dynamic scheduling on video transcoding for MPEG DASH in the cloud environment," in *Proceedings of the 5th ACM Multimedia Systems Conference on - MMSys '14*. Singapore, Singapore: ACM Press, 2014, pp. 283–294. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2557642.2557656

[76] N. Barman, N. Khan, and M. G. Martini, "Analysis of Spatial and Temporal Information Variation for 10-Bit and 8-Bit Video Sequences," in *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. Limassol, Cyprus: IEEE, Sep. 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8858486/

[77] P. Agrawal, A. Zabrovskiy, A. Ilangovan, C. Timmerer, and R. Prodan, "FastTTPS: fast approach for video transcoding time prediction and scheduling for HTTP adaptive streaming videos," *Cluster Computing*, vol. 24, no. 3, pp. 1605–1621, Sep. 2021. [Online]. Available: https://link.springer.com/10.1007/s10586-020-03207-x

[78] H. Zhao, Q. Zheng, W. Zhang, and J. Wang, "Prediction-Based and Locality-Aware Task Scheduling for Parallelizing Video Transcoding Over Heterogeneous MapReduce Cluster," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 4, pp. 1009–1020, Apr. 2018. [Online]. Available: http://ieeexplore.ieee.org/document/7763776/

[79] P. Paakkonen, A. Heikkinen, and T. Aihkisalo, "Architecture for Predicting Live Video Transcoding Performance on Docker Containers," in *2018 IEEE International Conference on Services Computing (SCC)*. San Francisco, CA, USA: IEEE, Jul. 2018, pp. 65–72. [Online]. Available: https://ieeexplore.ieee.org/document/8456402/

[80] P. Pääkkönen, A. Heikkinen, and T. Aihkisalo, "Online architecture for predicting live video transcoding resources," *Journal of Cloud Computing*, vol. 8, no. 1, p. 9, Dec. 2019. [Online]. Available: https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-019-0132-0

[81] O. Barais, J. Bourcier, Y.-D. Bromberg, and C. Dion, "Towards microservices architecture to transcode videos in the large at low costs," in *2016 International Conference on Telecommunications and Multimedia (TEMU)*. Heraklion, Crete, Greece: IEEE, Jul. 2016, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7551918/

[82] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang, and Y. Gao, "Container Based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. San Francisco, CA, USA: IEEE, Jun. 2016, pp. 758–765. [Online]. Available: http://ieeexplore.ieee.org/document/7820342/

[83] L. Van Ma, J. Park, J. Nam, J. Jang, and J. Kim, "An efficient scheduling multimedia transcoding method for DASH streaming in cloud environment," *Cluster Computing*, vol. 22, no. S1, pp. 1043–1053, Jan. 2019. [Online]. Available: http://link.springer.com/10.1007/s10586-017-1259-8

[84] Y. Dong, X. Zhang, Y. Zhao, and L. Song, "A containerized media cloud for video transcoding service," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*. Las Vegas, NV: IEEE, Jan. 2018, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/8326347/

[85] L. De Cicco, S. Mascolo, and V. Palmisano, "Feedback control for adaptive live video streaming," in *Proceedings of the second annual ACM conference on Multimedia*

systems. San Jose CA USA: ACM, Feb. 2011, pp. 145–156. [Online]. Available: https://dl.acm.org/doi/10.1145/1943552.1943573

[86] A. Erfanian, H. Amirpour, F. Tashtarian, C. Timmerer, and H. Hellwagner, "LwTE-Live: Light-weight Transcoding at the Edge for Live Streaming," in *Proceedings of the Workshop on Design, Deployment, and Evaluation of Network-assisted Video Streaming*. Virtual Event Germany: ACM, Dec. 2021, pp. 22–28. [Online]. Available: https://dl.acm.org/doi/10.1145/3488662.3493829

[87] H. Mao, R. Netravali, and M. Alizadeh, "Neural Adaptive Video Streaming with Pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Los Angeles CA USA: ACM, Aug. 2017, pp. 197–210. [Online]. Available: https://dl.acm.org/doi/10.1145/3098822.3098843

[88] Z. Akhtar, Y. S. Nam, R. Govindan, S. Rao, J. Chen, E. Katz-Bassett, B. Ribeiro, J. Zhan, and H. Zhang, "Oboe: auto-tuning video ABR algorithms to network conditions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. Budapest Hungary: ACM, Aug. 2018, pp. 44–58. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230558

[89] L. Xie, C. Zhou, X. Zhang, and Z. Guo, "Dynamic threshold based rate adaptation for HTTP live streaming," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. Baltimore, MD, USA: IEEE, May 2017, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/8050574/

[90] J. Vanne, M. Viitanen, T. D. Hamalainen, and A. Hallapuro, "Comparative Rate-Distortion-Complexity Analysis of HEVC and AVC Video Codecs," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1885–1898, Dec. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6324420/

[91] Jiajia Lu, Fan Liang, Liwei Xie, and Yonglin Luo, "A fast block partition algorithm for HEVC," in *2013 9th International Conference on Information, Communications & Signal Processing*. Tainan, Taiwan: IEEE, Dec. 2013, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/document/6782918/

[92] "Command line options — x265 documentation," https://x265.readthedocs.io/en/2.5/cli.html.

[93] "Threading — x265 documentation." [Online]. Available: https://x265.readthedocs.io/en/stable/threading.html

[94] M. Alvarez-Mesa, C. C. Chi, B. Juurlink, V. George, and T. Schierl, "Parallel video decoding in the emerging HEVC standard," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Kyoto, Japan: IEEE, Mar. 2012, pp. 1545–1548. [Online]. Available: https://ieeexplore.ieee.org/document/6288186/

[95] 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, Dec. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6547985/

[96] "Command line options — x265 documentation." [Online]. Available: https://x265.readthedocs.io/en/stable/cli.html

[97] "Preset options — x265 documentation." [Online]. Available: https://x265.readthedocs.io/en/stable/presets.html

[98] Y.-C. Lin, H. Denman, and A. Kokaram, "Multipass encoding for reducing pulsing artifacts in cloud based video transcoding," in *2015 IEEE International Conference on Image Processing (ICIP)*. Quebec City, QC, Canada: IEEE, Sep. 2015, pp. 907–911. [Online]. Available: http://ieeexplore.ieee.org/document/7350931/

[99] "CRF Guide (Constant Rate Factor in x264, x265 and libvpx)." [Online]. Available: https://slhck.info/video/2017/02/24/crf-guide.html

[100] B. B. Bunny. [Online]. Available: https://peach.blender.org/

[101] Xiph, https://media.xiph.org/video/derf/.

[102] A. W. Services, "Aws elemental." [Online]. Available: https://www.elemental.com/resources/4k-test-sequences

[103] "(2) aws for media & entertainment - youtube." [Online]. Available: https://www.youtube.com/channel/UCj7FkILntTrAzITWw4MWEFA

[104] "Bind mounts — docker documentation." [Online]. Available: https://docs.docker.com/storage/bind-mounts/

[105] A. Mercat, M. Viitanen, and J. Vanne, "UVG dataset: 50/120fps 4K sequences for video codec analysis and development," in *Proceedings of the 11th ACM Multimedia Systems Conference*. Istanbul Turkey: ACM, May 2020, pp. 297–302. [Online]. Available: https://dl.acm.org/doi/10.1145/3339825.3394937

[106] N. Bouzakaria, C. Concolato, and J. Le Feuvre, "Overhead and performance of low latency live streaming using MPEG-DASH," in *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications*. Chania, Crete, Greece: IEEE, Jul. 2014, pp. 92–97. [Online]. Available: http://ieeexplore.ieee.org/document/6878732/

[107] W. Robitza, "SITI: Spatial Information / Temporal Information," Dec. 2021, original-date: 2017-10-24T11:41:16Z. [Online]. Available: https://github.com/slhck/siti

[108] "Rapid access cloud – cybera." [Online]. Available: \url{https://www.cybera.ca/rapid-access-cloud/}

[109] "Keras: the python deep learning api." [Online]. Available: https://keras.io/

[110] "Cloud automl custom machine learning models — google cloud." [Online]. Available: https://cloud.google.com/automl

[111] X. He, K. Zhao, and X. Chu, "AutoML: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, Jan. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950705120307516

[112] "About data splits for automl models — vertex ai — google cloud." [Online]. Available: https://cloud.google.com/vertex-ai/docs/general/ml-use

[113] "Training vs testing vs validation sets — towards data science." [Online]. Available: https://towardsdatascience.com/training-vs-testing-vs-validation-sets-a44bed52a0e1

[114] "Rate models — automl tables — google cloud." [Online]. Available: https://cloud.google.com/automl-tables/docs/evaluate?hl=fr

[115] "A gentle introduction to ensemble learning algorithms." [Online]. Available: https://machinelearningmastery.com/tour-of-ensemble-learning-algorithms/

[116] "View model architecture with cloud logging — automl tables — google cloud." [Online]. Available: https://cloud.google.com/automl-tables/docs/logging?hl=fr

[117] "aliasif1/vertex_ai_transcoding_time_model: Model returned by google vertex ai for predicting the transcoding time." [Online]. Available: https://github.com/aliasif1/vertex_ai_transcoding_time_model

[118] "Accuracy vs. f1-score. a comparison between accuracy and... — by purva huilgol — analytics vidhya — medium." [Online]. Available: https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2

[119] "Vertex ai cpu cores prediction model - google drive." [Online]. Available: https://drive.google.com/drive/folders/1jViPw0fgmnhX3RzdO7s3ZGet6c2HuIoy

[120] Y. Sun, L. Li, Z. Li, and S. Liu, "Yoco: Light-Weight Rate Control Model Learning," p. 5.

[121] M. Covell, M. Arjovsky, Y.-c. Lin, and A. Kokaram, "Optimizing Transcoder Quality Targets Using a Neural Network with an Embedded Bitrate Model," *Electronic Imaging*, vol. 28, no. 2, pp. 1–7, Feb. 2016. [Online]. Available: https://library.imaging.org/ei/articles/28/2/art00016

[122] B. Cheng and Y. Zhang, "A neural network approach to GOP-level rate control of x265 using Lookahead," Oct. 2019, arXiv:1908.02939 [eess]. [Online]. Available: http://arxiv.org/abs/1908.02939

[123] H. Xing, Z. Zhou, J. Wang, H. Shen, D. He, and F. Li, "Predicting Rate Control Target Through A Learning Based Content Adaptive Model," in *2019 Picture Coding Symposium (PCS)*. Ningbo, China: IEEE, Nov. 2019, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/8954541/

[124] M. Utke, S. Zadtootaghaj, S. Schmidt, S. Bosse, and S. Möller, "NDNetGaming - development of a no-reference deep CNN for gaming video quality prediction," *Multimedia Tools and Applications*, vol. 81, no. 3, pp. 3181–3203, Jan. 2022. [Online]. Available: https://link.springer.com/10.1007/s11042-020-09144-6

[125] "Kubernetes." [Online]. Available: https://kubernetes.io/

[126] "Swarm mode overview — docker documentation." [Online]. Available: https://docs.docker.com/engine/swarm/

[127] S. Arora and J. Kramer, "Lecture 17: Decision Tree Complexity." [Online]. Available: https://users.cs.duke.edu/~reif/courses/complectures/Arora/lec17.pdf

[128] "Dropout in neural networks - geeksforgeeks." [Online]. Available: https://www.geeksforgeeks.org/dropout-in-neural-networks/#:~:text=In

Appendix A

Machine Learning Models

A.1 Vertex AI Models

Once a model is trained by Google AutoMl (Vertex AI), it provides information on the final model along with the models it used during the trial phase.

For the Boosted Trees, some of the important architecture parameters logged by AutoMl includes maximum tree depth, number of trees, tree complexity, l1 regularization and l2 regularization. A brief summary of these parameters is as follows:

- Maximum Tree Depth: The highest number of nodes (decision points) along a tree path
- Number of Trees: The number of trees constituting the boosted model
- Tree Complexity: The tree complexity for worst case inputs [127]
- L1 Regularization (Lasso Regularization): Introduced to avoid model overfitting
- L2 Regularization(Ridge Regularization): Introduced to avoid model overfitting

For the Neural Networks, some of the important architecture parameters logged by AutoMl includes dropout rate, hidden layer size, number of cross layers, and number of hidden layers. A brief summary of these parameters is as follows:

Table A.1: Architecture of the Neural Network constituting the final Transcoding time prediction model

S.N	Model	Dropout	Hidden Layer Size	# Cross Layers	# Hidden Layers
1	Neural Network	0.125	64	0	2
2	Neural Network	0.125	64	0	2
3	Neural Network	0.125	64	0	2
4	Neural Network	0.125	64	0	2
5	Neural Network	0.125	64	0	2
6	Neural Network	0	16	0	4
7	Neural Network	0	16	0	4
8	Neural Network	0	16	0	4
9	Neural Network	0	16	0	4
10	Neural Network	0	16	0	4

- Dropout: The proportion of neurons which are turned off (removed from the neural network) [128]
- Hidden Layer Size: The number of neurons in the hidden layer.
- Number of Hidden Layers: The number of layers between the input layer and the output layer.

Table A.1 describes the architecture of the 10 Neural Networks which constitute the final model for predicting the transcoding time. Table A.2 describes the architecture of the 15 Boosted Trees which constitute the final model for predicting the transcoding time.

For predicting the CPU cores, the final model returned by AutoMl consists of multiple models. Specifically, the final model consists of 25 neural networks. Table A.3 describes the architecture of the Neural Networks which constitute the final model for predicting the transcoding time.

A more detailed architecture of the transcoding time prediction model can be found at [117]. Similarly, a detailed architecture of the number of CPU cores prediction model can be found at [119]

Table A.2: Architecture of the Boosted Trees constituting the final Transcoding time prediction model

S.N	Model	L1	L2	Max Tree Depth	# Trees	Tree Complexity
1	Boosted Trees	1	3	9	100	0.1
2	Boosted Trees	1	3	9	100	0.1
3	Boosted Trees	1	3	9	100	0.1
4	Boosted Trees	1	3	9	100	0.1
5	Boosted Trees	1	3	9	100	0.1
6	Boosted Trees	0.1	1	9	350	0.1
7	Boosted Trees	0.1	1	9	350	0.1
8	Boosted Trees	0.1	1	9	350	0.1
9	Boosted Trees	0.1	1	9	350	0.1
10	Boosted Trees	0.1	1	9	350	0.1
11	Boosted Trees	0.3	1	9	50	0.1
12	Boosted Trees	0.3	1	9	50	0.1
13	Boosted Trees	0.3	1	9	50	0.1
14	Boosted Trees	0.3	1	9	50	0.1
15	Boosted Trees	0.3	1	9	50	0.1

A.2 Keras Models

To begin with we used `python` and `keras` [109] and developed manual ANN models for the two prediction tasks *i.e.,* for predicting the transcoding time and for predicting the number of CPU cores needed for a transcoding task. The models are summarized in the following subsections.

A.2.1 Keras Transcoding time Prediction Model

We used 11 features to train the transcoding time prediction model - duration, fps, input height, output height, input pixels, output pixels, downscaled SI, downscaled TI, downscaled bitrate, preset and the number of CPU cores allocated to the transcoder. We decided to go with a ANN as the model type as suggested by [23, 77]. The model is a regression model and the parameters of the model pipeline are presented in Table A.4. Table A.5 shows the performance metrics of the model

Table A.3: Architecture of the Neural Networks constituting the final CPU cores prediction model

S.N	Model	Dropout	Hidden Layer Size	# Cross Layers	# Hidden Layers
1	Neural Network	0.125	128	0	3
2	Neural Network	0.125	128	0	3
3	Neural Network	0.125	128	0	3
4	Neural Network	0.125	128	0	3
5	Neural Network	0.125	128	0	3
6	Neural Network	0.125	64	0	4
7	Neural Network	0.125	64	0	4
8	Neural Network	0.125	64	0	4
9	Neural Network	0.125	64	0	4
10	Neural Network	0.125	64	0	4
11	Neural Network	0.25	1024	0	4
12	Neural Network	0.25	1024	0	4
13	Neural Network	0.25	1024	0	4
14	Neural Network	0.25	1024	0	4
15	Neural Network	0.25	1024	0	4
16	Neural Network	0.25	1024	0	2
17	Neural Network	0.25	1024	0	2
18	Neural Network	0.25	1024	0	2
19	Neural Network	0.25	1024	0	2
20	Neural Network	0.25	1024	0	2
21	Neural Network	0	1024	0	2
22	Neural Network	0	1024	0	2
23	Neural Network	0	1024	0	2
24	Neural Network	0	1024	0	2
25	Neural Network	0	1024	0	2

Table A.4: Keras model parameters for the transcoding time prediction model

Parameter	Value
Number of hidden layers	2
Number of neurons in the hidden layer	64,64
Loss function	Mean Absolute Error
Batch size	64
Epochs	3200
Train:Test size	80:20

Table A.5: Performance Metrics for the Keras transcoding time prediction model

Metric	Value
Mean Absolute Error	0.399 seconds
Mean Squared Error	2.32
Mean Absolute Percentage Error	5.36 %

Table A.6: Keras model parameters for the number of CPU cores prediction model

Parameter	Value
Number of hidden layers	3
Number of neurons in the hidden layer	128,128,128
Loss function	Categorical Cross-entropy
Batch size	64
Epochs	1600
Train:Test size	80:20

A.2.2 Keras CPU cores Prediction Model

We used 11 features to train the number of CPU cores prediction model - duration, fps, input height, output height, input pixels, output pixels, downscaled SI, downscaled TI, downscaled bitrate, preset and the transcoding time. We decided to go with a ANN as the model type. The model is a classification model and the parameters of the model pipeline are presented in Table A.6. Table A.7 shows the performance metrics of the model

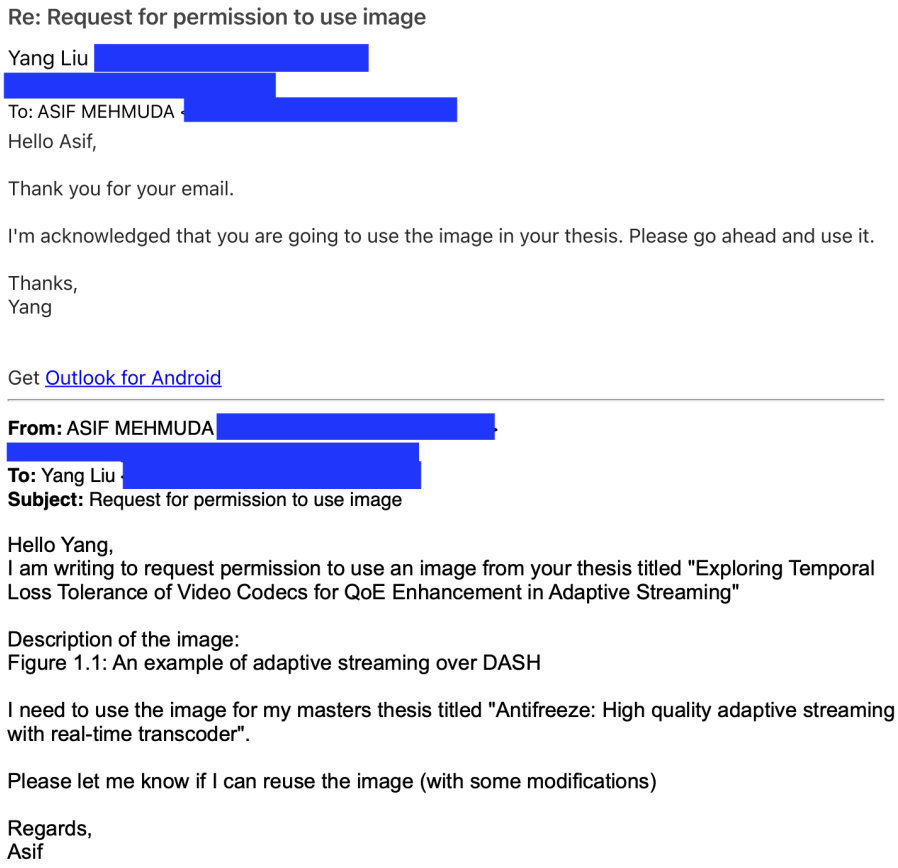
Table A.7: Performance Metrics for the Keras Number of CPU cores prediction model

Metric	Value
Accuracy	97 %
Precision	97 %
Recall	0.97

Appendix B

Copyright Permissions

Fig. 2.2 was inspired by the work published in [2].



Re: Request for permission to use image

Yang Liu [redacted]

To: ASIF MEHMUDA [redacted]

Hello Asif,

Thank you for your email.

I'm acknowledged that you are going to use the image in your thesis. Please go ahead and use it.

Thanks,
Yang

Get Outlook for Android

From: ASIF MEHMUDA [redacted]

To: Yang Liu [redacted]
Subject: Request for permission to use image

Hello Yang,
I am writing to request permission to use an image from your thesis titled "Exploring Temporal Loss Tolerance of Video Codecs for QoE Enhancement in Adaptive Streaming"

Description of the image:
Figure 1.1: An example of adaptive streaming over DASH

I need to use the image for my masters thesis titled "Antifreeze: High quality adaptive streaming with real-time transcoder".

Please let me know if I can reuse the image (with some modifications)

Regards,
Asif

Figure B.1: Copyright Permission Email Screenshot