

CONTRAST: Container-based Transcoding for Interactive Video Streaming

Sajad Sameti
Department of Electrical and
Computer Engineering
University of Calgary
Calgary, Alberta
Email: sajad.sameti@ucalgary.ca

Mea Wang
Department of Computer Science
University of Calgary
Calgary, Alberta
Email: meawang@ucalgary.ca

Diwakar Krishnamurthy
Department of Electrical and
Computer Engineering
University of Calgary
Calgary, Alberta
Email: dkrishna@ucalgary.ca

Abstract—Interactive video streaming applications are becoming increasingly popular. To maintain the Quality of Experience (QoE) of an end user, interactive streaming platforms need to transcode a video stream, *i.e.*, adapt the quality of the video content, to match the network conditions between the platform and the user as well as the device capabilities of the end user. Modern video codecs such as High Efficiency Video Coding (HEVC) require significant computational resources for transcoding operations. Consequently, there is a need for systems that can perform transcoding quickly at runtime to sustain the real-time performance required for interactive streaming while at the same time using just the right amount of computational resources for the transcoding operations. This paper addresses this need by designing and implementing CONTRAST, a Container-based Distributed Transcoding Framework for Interactive Video Streaming. For any given stream and transcoding resolution, CONTRAST exploits a profiling technique to automatically determine the degree of parallelism, *i.e.*, the number of processing cores, demanded by the transcoding process to sustain the stream's frame rate. It then launches Docker containers configured with the required number of cores to perform the transcoding. Experiments using a set of realistic video streams show that CONTRAST is able to sustain the frame rate requirements for interactive streams in a more resource efficient manner compared to baseline techniques that do not consider the degree of parallelism. To the best of our knowledge, our paper is the first to establish best practices for implementing transcoding platforms for interactive streaming videos encoded using a modern video codec.

I. INTRODUCTION

Network-based video services continue to be popular. According to Cisco, 82% of all Internet traffic will be video by 2020 [8]. In particular, a recent trend has been the increase in popularity of interactive streaming services, which host live video streams. For example, the Twitch.tv [20] live streaming video platform records more than 50 million viewers per month [5].

Users of video services have different devices and experience varied network conditions, thus the specification of content they produce and consume is different. For example devices with high processing power and higher resolution screens are able to play higher quality videos, while others may need videos in a different resolution or less complex codec. Furthermore, more powerful devices with higher quality cameras can record videos at higher frame rates and qualities.

In order to provide acceptable Quality of Experience (QoE) to all users, video content should be transcoded to match a user's requirements [13]. Video transcoding is a process in which different characteristics of a video such as resolution, bitrate and playback complexity can be changed. In classical applications such as Video On Demand (VOD), videos are transcoded into multiple versions characterized by different resolutions and bitrates. Using protocols such as Dynamic Adaptive Streaming over HTTP (DASH), a user can request the version that best fits their device capabilities and network conditions.

Transcoding presents considerable challenges in an interactive streaming environment. In contrast to a classical VOD service where whole videos can be created and stored in advance, in interactive streaming the video content is being created as it is being consumed. Consequently, transcoding has to be performed on the fly to satisfy user QoE requirements. Unfortunately, transcoding is a very computationally intensive task. Large transcoding delays can disrupt the real-time nature of a transcoded stream and hence a user's QoE by causing the rate at which video frames are transcoded, *i.e.*, the transcoding throughput, to lag the frame rate demanded by the stream [12]. For this reason, Twitch.tv only transcodes videos for their premium broadcasters with the rest of the content being streamed in the original format [5]. This motivates the need for intelligent transcoding techniques that can sustain the real-time performance requirements of interactive streaming sessions.

Devising transcoding techniques that can sustain interactive streams while simultaneously making efficient use of compute resources remains an open problem. A number of recent studies have exploited distributed computing to speedup the transcoding process [14], [18], [10], [16]. However, these studies consider stored videos and do not focus on meeting real-time performance requirements. This paper aims to address this gap in literature.

We propose CONTRAST, a container-based distributed transcoding framework for interactive video streaming. CONTRAST uses an offline profiling technique to automatically infer the degree of parallelism, *i.e.* the number of processing cores, needed to sustain the frame rate of a transcoded stream for any given transcoding requirement and transcoding hardware platform. CONTRAST exploits the fine-grained resource allocation mechanisms supported by Docker containers [2]

to perform the transcoding using the deduced degree of parallelism.

Experimental results using realistic video streams show that CONTRAST can sustain real-time streaming in a resource efficient manner. In particular, CONTRAST exploits our empirical observation that the degree of parallelism that can benefit a stream depends only on the source and target resolutions of the stream. Increases beyond this limit are unlikely to yield performance gains. Using this observation and the resource allocation mechanisms supported by Docker technology, CONTRAST is able to right-size the transcoding process. The paper also provides a detailed sensitivity analysis of factors that can influence the performance of CONTRAST.

Results also show that CONTRAST is more resource efficient in meeting real-time performance when compared to GPU and native approaches that do not permit fine-grained control over the resources allocated to a transcoding operation. Consider Amazon EC2 pricing [1], our experiments indicate that the GPU approach is 6 times more expensive than CONTRAST.

II. RELATED WORK

Video transcoding has been identified as one of the limiting factors for QoE in applications such as AR/VR that involve interactive video streaming [6], [4]. To address this challenge, Braitmaier *et al.* proposed an edge solution to transcode user-generated videos before transmission to the cloud and other users in an interactive streaming session. Bilal *et al.* proposed a similar edge computing solution to offload storage and bandwidth demands placed on the core cloud by interactive streaming sessions.

Along with edge solutions to reduce the workload of the core cloud, it is also essential to speedup the transcoding process. As the transcoding standards advance from H.261 to H.265 (HEVC - the latest standard), compression, visual quality restoration, and transcoding techniques have all improved significantly. Despite these advances, transcoding is still a complex and compute-intensive process. Several studies have proposed systems that improve throughput by processing multiple videos concurrently on a cluster of compute nodes [14], [18]. While these techniques increase the number of videos transcoded at any given time, they do not focus on speeding up the transcoding time of individual videos. In contrast, other studies propose solutions [10], [16] where multiple segments belonging to a video are processed in parallel on different nodes to speedup the overall transcoding time of that video. In contrast to our work, these solutions focus on stored videos where all segments of a video are available prior to transcoding. Achieving real-time performance is challenging in interacting streaming systems since at any given time only a small number of segments of a video are available for transcoding.

Containers have been introduced as an alternative to virtual machines (VM) to virtualize computing resources. Unlike VMs where hardware resources are virtualized and shared, a container is a lightweight process that bundles application and

necessary supports (*e.g.*, installations and configurations) that can be dropped into a given operating system for execution. Containers are very fast to deploy and destroy. Docker is an open source technology that facilitates managing containers through APIs and Command Line Interface (CLI) tools. Docker runs a virtualization layer called Docker engine that manages multiple containers. It offers fine-grained resource allocation for individual containers. For example, it is possible to allocate fractions of CPUs (*i.e.*, as little as 1% of CPU) to a container. As we describe later, we exploit this feature to realize resource-efficient real-time transcoding.

Containers have been used in variety of applications for fine-grained resource allocation and management. Several studies have shown that Docker containers can achieve near native performance [7], [22]. Dong *et al.* design a containerized system for processing videos [9]. Sameti *et al.* [17] outline a system architecture that integrates transcoding with containers. However, in contrast to our work, that study does not investigate resource allocation techniques needed for a containerized system to achieve real-time performance while simultaneously making use of resources in an efficient manner.

Despite the benefits of containers, we are not aware of existing work that provides best practices for exploiting containers for real-time transcoding. We address this gap with this work.

III. AN OVERVIEW OF VIDEO TRANSCODING

The speedup of distributed video transcoding of any given video codec standard is limited by the performance of cores executing the process and the degree of parallelism of the transcoding process. The former determines the speedup of each branch of the parallel process. The performance of cores varies from one deployment to another and is essentially determined by hardware specifications. The latter aspect determines the maximum number of cores that the process will need for the most speedup, *i.e.*, more cores beyond this maximum number will not offer further speedup. This factor typically does not vary regardless of the specifications of the cores, but may vary for different codec standards. Thus, in order to achieve the maximum speedup in any deployment and to be resource efficient, it is important to know the degree of parallelism of the transcoding process. In this section, we present a qualitative analysis of factors that influence the degree of parallelism. We exploit this analysis in the design of CONTRAST.

A video is essentially a sequence of segments that consists of one or more Group of Pictures (GOPs). Coding standards (*e.g.*, H.264/AVC and H.265/HEVC) define how frames are encoded into GOPs, each consisting of a number of coded frames, as illustrated in Fig. 1. Each GOP of the same video represents the same playback length (typically 1 second) and consists of the same number of frames. Thus, each frame has the same playback length. The frame rate is defined as the number of frames constituting one second of playback, measured as frames-per-second (fps). Typical frame rates are 30 fps, 60 fps, 90 fps, and 120 fps. The higher the frame rate, the smoother the motion. These coded frames could be

of type I, P or B, with the I-frame indicating the beginning of a GOP. For video streaming, the GOP structure is often closed, meaning each GOP is independent of other GOPs so that they can be transmitted and processed separately. Within a GOP, the I-frames are independent. Each P-frame within a GOP depends on the preceding I-frame or P-frame, and B-frames depend on the I-frame and P-frames. Therefore, we focus only on the degree the parallelism within a GOP required to achieve real-time video transcoding for interactive video streaming.

Since closed GOPs are independent of each other, GOPs can be transcoded in parallel. Thus, the degree of parallelism of a segment is $D \times S$ where D is the degree of parallelism of a single GOP and S is the number of GOPs in a segment. In most interactive streaming applications, a segment consists of only 1 GOP and one GOP usually represents 1 playback second. Without loss of generality, in this paper we follow the same convention and let D denote the degree of parallelism of a GOP as well as a segment.

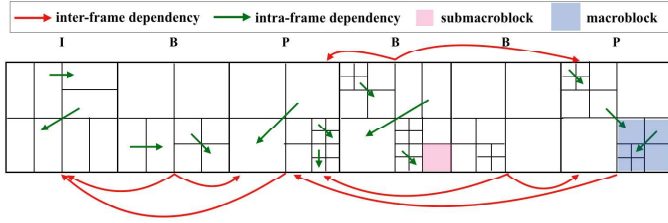


Figure 1. Structure of a GOP

As illustrated in Fig. 1, each frame is further divided into macroblocks and submacroblocks for processing. In the H.265 video coding standard family, macroblocks are rectangular blocks of 32×32 or 64×64 pixels inside each frame. Each macroblock may be divided to rectangular submacroblocks 4×4 , 4×8 , 8×4 , 8×8 , 8×16 or 16×8 pixels. Both the macroblock and the submacroblock are the basic processing units during the transcoding process, and we will collectively refer to both terms as blocks henceforth for simplicity. In the most recent HEVC standard, macroblocks and submacroblocks are referred to as Coding Tree Unit (CTU) and Coding Units (CU), respectively. A consecutive set of CTUs forms a row. Processing of a frame starts from the top left and moves forward in rows. Inside each CTU, the CUs are processed in a “Z” fashion.

There are two types of dependencies among blocks namely, spatial and temporal. The spatial dependency or intra-frame dependency refers to the dependency among blocks within a frame. The temporal dependency or inter-frame dependency refers to the dependency among blocks across frames. Thus, there are two layers of parallelism namely, processing multiple blocks within a frame in parallel and processing multiple frames in parallel. In either layer, the blocks must be processed following their chain of dependency, and only blocks that are independent of each other can be processed in parallel. Thus, the number of independent chains is the degree of parallelism of the transcoding process. To the best of our knowledge, due to the complexity of block structures and dependency among blocks, there is no simple way to determine

the degree of parallelism. For example, in HEVC in order to start processing a row in a frame its reference frame should be at least be two rows ahead. Moreover, in order to process a CTU, the row above that CTU should be three CTUs ahead in processing. However, it is intuitive that the more blocks in a frame, the higher the degree of intra-frame and inter-frame parallelism. Thus, the degree of parallelism is directly related to the resolution of the video. In the next section, we will propose a systematic approach to determine this key measure as part of our distributed transcoding scheme.

IV. CONTRAST

The fine granularity of resource allocation offered by containers motivate us to propose CONTRAST. In a nutshell, the CONTRAST framework has two parts: an offline profiler and a real-time distributed transcoder. The offline profiler determines the degree of parallelism and achievable per-segment transcoding throughput (in fps) on the (physical or virtual) machine(s) where the video will be transcoded. The real-time transcoding scheduler determines the number of cores needed to achieve overall transcoding throughput matching the video frame rate to sustain an interactive streaming session, allocates appropriate number of cores, and launches containerized transcoders on these cores. The design of each of the two parts is detailed in the rest of this section.

A. Offline Profiler

As discussed in Sec. III, the speedup of distributed video transcoding of any given video codec standard is limited by the performance of cores executing the process and the degree of parallelism of the transcoding process. The former is directly related to the speed of the cores, while the latter is directly related to the video resolution and the codec standard in use. Since both factors are independent of the actual video content, we design an offline profiler to profile the codec standard on a set of benchmark cores.

The profiler takes two inputs: the transcoder of the selected codec standard and a highest resolution sample (short) video in the selected codec standard. The profiler executes the transcoder to transcode the sample video in a range of source and target resolutions. For example, since typical resolutions are 4K (Ultra High Definition - UHD), 1080p (Full High Definition - FullHD), 720p (High Definition - HD), and 480p (Standard Definition - SD), for a 4K sample video, the profiler will profile the following transcoding combinations: $4K \rightarrow 1080p$, $4K \rightarrow 720p$, $4K \rightarrow 480p$, $1080p \rightarrow 720p$, $1080p \rightarrow 480p$, and $720p \rightarrow 480p$. For each setting, the profiler executes the transcoder to transcode the sample video on increasing numbers of cores until the speedup converges. This converging point is the degree of parallelism $D_{i \rightarrow j}$, where i is the source resolution and j is the target resolution. The degree of parallelism for all settings are presented by set $\mathbf{D} = \{D_{i \rightarrow j}\}$. Along with the degree of parallelism, the profiler also records the achievable per-segment transcoding throughput for each setting $\mathbf{T}^b = \{T_{i \rightarrow j}^b\}$, measured in frame rate (fps), on the (physical or virtual) benchmark machine

where the profiling is performed. Both $\mathbf{D}$ and $\mathbf{T}^b$ will be input to the real-time distributed transcoding scheduler.

The profiling cannot always be performed on cores where the transcoding will be performed. For instance, the cores assigned to the transcoding task (also referred to as target cores) may be dynamically determined at runtime due to various reasons. In this case, we need to use the profiling data gathered on the benchmarking machine to estimate the degree of parallelism $\mathbf{D}$ and the transcoding throughputs for various settings on the target machine. While the degree of parallelism $\mathbf{D}$ is independent of the machine, the transcoding throughput $\mathbf{T}$ may scale up and down depending on the relative speeds of the benchmark and target machines. Since transcoding is a CPU-intensive operation, we execute a standard CPU micro-benchmark on both the benchmark and target machines to determine a performance scaling factor α_k . The scaling factor is a real number $\alpha_k > 0$, where $0 < \alpha_k < 1$ implies scaling down (speed down) and $\alpha_k > 1$ implies scaling up (speed up). The transcoding throughput $T_{i \rightarrow j}$ on the target machine is estimated by the real-time distributed transcoding scheduler to be $T_{i \rightarrow j}^b \times \alpha_k$. The scaling factors for various target machines are presented by the set $\alpha = \{\alpha_k\}$, which will also be passed to the real-time distributed transcoding scheduler.

B. Real-time Distributed Transcoding Scheduler

The real-time distributed transcoding scheduler performs the following tasks: First, it determines the number of cores needed to achieve transcoding throughput matching the video frame rate to sustain an interactive streaming session. It then allocates the cores on target physical or virtual machines. At last, it launches containerized transcoders on the target cores. Each containerized transcoder is responsible for one segment and is allocated $D_{i \rightarrow j}$ cores for the best speedup of the transcoding process. Assume that the job is to transcode a video from resolution i to resolution j . The number of cores is determined based on the degree of parallelism of the job $D_{i \rightarrow j}$ (i.e., the maximum number of cores needed to transcode a single segment), the transcoding throughput of the job on the target cores $T_{i \rightarrow j}$, and the video frame rate R . If $T_{i \rightarrow j} > R$, then the job only needs $N = D_{i \rightarrow j}$ cores. Thus, the scheduler can schedule 1 GOP for transcoding at a time and outputs the coded frames at a rate equal to or higher than the video frame rate, which is sufficient to sustain the interactive streaming session. Otherwise, the scheduler needs to schedule $P = \lceil \frac{R}{T_{i \rightarrow j}} \rceil$ segments for parallel transcoding in order to produce coded frames at a rate meeting the video frame rate. The number of cores needed by this job would be $N = D_{i \rightarrow j} * P$.

Before launching P containerized transcoders to transcode P segment(s) in parallel, the scheduler selects cores on the machines with the objective to minimize the number of machines involved. Dedicating machines in their entirety as much as possible will avoid contention from other computational tasks that could be deployed on these machines to utilize the spare core capacity. This will reduce time spent on context switching and processor cache misses. The scheduler sorts the machines

by the number of available cores in descending order. It then starts allocating cores from the machine with the most number of cores until all cores are found to launch all P containerized transcoders.

V. IMPLEMENTATION

A. Offline Profiler

As discussed in Sec. IV-A, the profiler can be deployed on any machine since the degree of parallelism of a segment is independent of the hosting machine. We also introduced a mechanism to determine the performance scaling factor $\alpha_k > 0$ between the the benchmark machine and the target machine. In this implementation, we use a 8-server cluster as our benchmark machines. Each server has 16 2GHz cores and 16 GB of memory. All servers run Ubuntu 18.04.1 and Docker 18.06.1-ce. We use the “Big Buck Bunny” video with 30 fps frame rate for the profiling. From the 4K version of this video, we transcode the video to 4K, 1080p, 720p, and 480p. We also transcode the 1080p version to 720p and 480p and the 720p version to 480p, using the containerized transcoder. The detailed implementation of the containerized transcoder is presented in Sec. V-C. We profile the per-segment transcoding throughput $T_{i \rightarrow j}^b$ of each case on increasing number of cores and the results are shown in Fig. 2. From these results, we are also able to identify the degree of parallelism $D_{i \rightarrow j}$ (i.e., the minimum number of cores achieving the highest transcoding throughput) for each setting: 16 for 4K $\rightarrow$ 4K, 10 for 4K $\rightarrow$ 1080p, 9 for 4K $\rightarrow$ 720p, 8 for 4K $\rightarrow$ 480p, 8 for 1080p $\rightarrow$ 720p, 6 for 1080p $\rightarrow$ 480p, and 6 for 720p $\rightarrow$ 480p. This result validates our previous intuition that the degree of parallelism is dictated by the resolutions.

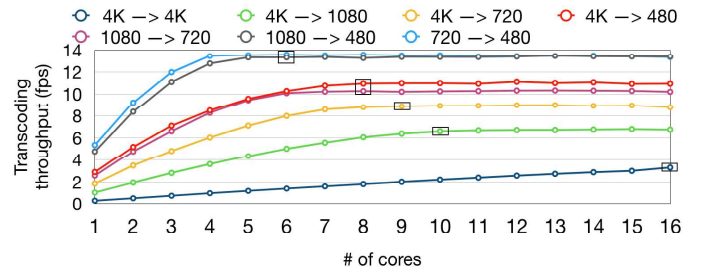


Figure 2. Profiling results for BBB video in the benchmark machines

To verify the mechanism to determine the performance scaling factor $\alpha_k > 0$ between the the benchmark machine and the target machine, we chose Amazon EC2 [1] *c5.4xlarge* instances with 16 vCPUs and 32GB memory. We ran the Sysbench [15] micro-benchmark tool on both the benchmark machines and EC2 instances. This tool has built-in tests to measure the CPU performance of the system. Since video transcoding is a CPU bound process measuring the speed of the CPU is enough to compare the transcoding speed of two machines. To measure the performance on both machines, we launched the profiler to profile individual cores. We ran 10000 CPU events and measured the speed of the CPU core in events per second. The scaling factor from the benchmarking machine to the EC2 *c5.4xlarge* instance α_{EC2} is found to be 3.6.

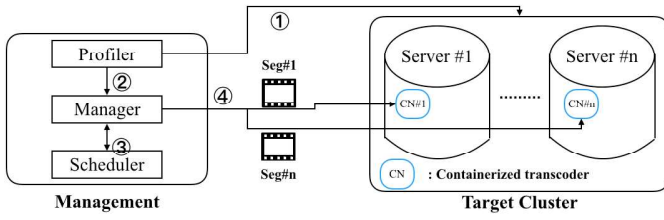


Figure 3. System Diagram

B. System Workflow

As a proof of concept and also for evaluation purpose, we implemented a distributed transcoding system based on the proposed CONTRAST framework. The offline profiling to measure $\mathbf{D}$ and $\mathbf{T}^b$ is done before the deployment. Deploying this solution is a two step process:

1. Cluster profiling: This step is conducted in order to map the power of the *Benchmarking Cluster* to the *Target Cluster*. Using a CPU benchmark the profiler evaluates the power of a single core in the *Target Cluster*. We then scale the resource requirements by factor of $\alpha_{Target Cluster}$ ①.

2. Video transcoding: In this step the measurements and the profiling results are passed to the manager module and the system is ready to start the transcoding ②. One container will be launched for each video segment on the number of cores determined in the profiling stage.

CONTRAST is a distributed scheme for video transcoding. It is agonistic to the actual codec standard or transcoding technique in use. The only factor CONTRAST will consider from the transcoding process is the degree of parallelism. The manager module takes the degree of parallelism profile $\mathbf{D}$, benchmarking transcoding throughput profile $\mathbf{T}^b$, and the scaling factor $\alpha_{Target Cluster}$ from the profiler ②. It also takes as input the profile of target machines (e.g., number of cores). Before the streaming session begins, the manager estimates the transcoding throughput on target machines $\mathbf{T}$ by multiplying $\mathbf{T}^b$ by the scaling factor $\alpha_{Target Cluster}$. As the segments arrive from the video source, the manager passes the degree of parallelism profile $\mathbf{D}$, transcoding throughput profile $\mathbf{T}$, and the the profile of target machines to the real-time distributed transcoding scheduler to determine the appropriate allocation of cores on each machine and the number of containers to be launched on each machine ③. It then launches the containerized transcoders and passes individual segments to each container for transcoding ④.

C. Containerized Transcoder

In order to launch transcoding jobs with fine-grained resource control, we built a Docker image with all necessary tools to transcode videos. This image is built based on the official Ubuntu 18.04 image. We chose the HEVC codec for the transcoding standard. HEVC is designed specifically to deliver high quality videos with high compression efficiency. It is thus able to support UHD (ultra high definition) videos e.g, 8K and 4K videos. This codec is capable of delivering videos with roughly half the bitrate compared to AVC[21]. *libx265* is the library which implements HEVC coding standard and

it is accessible through various video processing tools. In our implementation we chose FFMPEG [3], an open source tool, to access *libx265*. FFMPEG supports all the features of the library and almost all variations of video transcoding.

To transcode a segment, a container is launched using our pre-built docker image. The container takes the segment and the transcoding specs such as target resolution, compression to speed ratio and parallelism mode (discussed shortly) as input. It then uses these specifications to transcode the segment using FFMPEG.

libx265 implements inter-frame parallelism and intra-frame parallelism to increase transcoding speed. In inter-frame parallelism, a number *frame-threads* are launched to process frames in parallel. With intra-frame parallelism, a pool of threads is launched to process blocks within each frame in parallel. The number of *frame-threads* and the size of the pool (number of threads inside the pool) is determined by FFMPEG based on the detected hardware. We have three parallelism options when transcoding a segment, (1) *all-on*, which has both inter-frame and intra-frame parallelism enabled. (2) *inter-only*, which only has inter-frame parallelism. (3) *intra-only*, which only has intra-frame parallelism.

In the deployment of containerized transcoders, we have the option of *dynamic* deployment and *static* deployment. In a *static* deployment, the containers are already launched and ready to transcode segments. The manager assigns one segment to each container and waits for the completion. Once the transcoding of the current segment is done, the next segment is assigned to the container. In the *dynamic* deployment, however, the manager launches a new container to transcode a segment. This container is destroyed after the transcoding is finished and a new container is launched for the next segment. Dynamic deployment provides more flexibility. For example, in case we need to change the configuration (e.g number of cores) of the containerized transcoders we can accomplish that conveniently on a per segment basis with no interruption to the system. In contrast, with a static deployment we have to halt the container, destroy it and launch a new one with the desired configuration.

Containerized transcoders can also be deployed in *numa-aware* and *numa-blind* modes. In servers with NUMA architecture, a set of cores along with high speed memories and caches are packed into a unit called a NUMA node. Data transfer between cores in a NUMA node is extremely fast. However, transferring data between NUMA nodes is expensive and slow. With Docker, one can restrict a container to the cores in a specific NUMA node to avoid such overheads. However, restricting the cores on which a container can execute could restrict the scheduling options for the OS scheduler thereby causing potential performance degradations. We explore this in detail in Sec. VI.

VI. EVALUATION

Our evaluation uses three videos from a dataset proposed by the MSU group [11]. These represent a wide variety of videos in terms of spatial and temporal complexity. Characteristics of these videos are shown in Table I. Since the results for the three videos are very similar, in most cases we use the BBB video to illustrate key results.

Table I
VIDEO DATASET

Sequence	Bitrate (Mbps)	Frame rate (fps)	Average Motion Vector
Big Buck Bunny (BBB)	26.93	30	1.22
Highfields	48.25	25	1.52
Butterflies	30.19	30	0.54

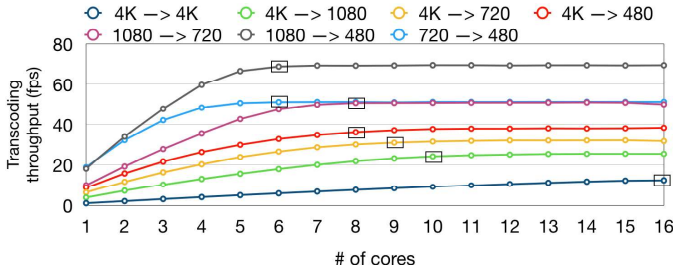


Figure 4. Profiling results for the BBB video in the EC2 machines

A. Profiler

Fig. 4 provides the measured profile for the degree of parallelism on EC2 for the BBB video. This result further confirms our earlier observation that $D_{i \rightarrow j}$ is solely determined by the source and target resolutions. For any given source-target resolution pair, the $D_{i \rightarrow j}$ value observed in this experiment matches that observed for our bare-metal setup as depicted in Fig. 2. We obtain the same values for $D_{i \rightarrow j}$ for the other videos in our dataset. Fig. 5 repeats our experiments for higher video frame rates for BBB namely, 60 fps and 120 fps. These results show that the $D_{i \rightarrow j}$ values do not depend on the video frame rate and can be predicted solely based on the source and target resolutions.

We next evaluate the accuracy of using profiling data collected on our bare-metal setup to estimate the transcoding throughput on our EC2 setup. The profiler suggests a scaling factor α_{EC2} of 3.6 for EC2. We use this value to estimate the transcoding throughput for various source and target resolution pairs at their appropriate $D_{i \rightarrow j}$ settings. From Table II, the estimated transcoding throughputs are very close to their corresponding measured values. Estimation errors are within 10% thereby validating our profiler-based core estimation approach.

B. Framework Tuning

Transcoder Parallelism: To evaluate the behaviour of parallelism modes in HEVC, we transcode different numbers of concurrent segments with intra-only, inter-only and all-on modes. The transcoding is performed using all 16 cores in one of the machines in our bare-metal setup.

Table II
PROFILER ACCURACY FOR $D_{i \rightarrow j}$ SETTINGS

Test Case	Bare-Metal	Prediction (EC2)	Measured (EC2)
4K To 4K	3.30	11.88	12.4
4K To 1080p	6.08	21.89	21.84
4K To 720p	8.58	30.90	28.70
4K To 480p	10.27	36.98	32.93
1080p To 720	11.65	41.94	42.85
1080p To 480p	15.93	57.35	59.65
720p To 480p	11.97	43.12	42.31

From Fig. 6, the all-on mode offers the best per-segment transcoding throughput. However, the per-segment throughput drops as the number of concurrent segments processed on the machine increases. This mode employs both frame and pool threads and hence incurs OS scheduling and context switching overheads as the number of concurrent segments increases. With intraframe parallel processing, macroblocks in each frame are transcoded from top left corner to the bottom right corner in a left-to-right and up-to-down manner. Due to the high-dependency among adjacent macroblocks, the speedup is not as significant as what interframe parallelism would offer. When it is enabled together with interframe parallel processing (all-on mode), only slight improvement in transcoding time is observed. The throughput of both all-on and inter-only modes become almost identical as the number of concurrent segments increases.

Comparing the CPU utilizations of all-on and inter-only, the inter-only approach has slightly lower utilizations and hence makes more efficient use of the cores. This suggests that the inter-only setting is a slightly better choice when multiple segments are being transcoded concurrently. The intra-only mode is not an ideal choice since it yields lower throughput than the other two approaches.

Container Management: We next compare the static and dynamic container management approaches outlined in Sec. V. We compare the performance of these two options while transcoding different numbers of concurrent segments in the BBB video. Each container transcodes from 4K to 1080p and is configured to use 8 cores. Fig. 7 shows the overall transcoding throughput across all segments for both approaches. Despite the overhead of redeploying containers, the dynamic approach has comparable performance to the static approach. Consequently, the dynamic approach is more preferable since it provides the flexibility of continuously adapting the number of cores assigned to a container to handle possible changes in transcoding requirements.

We next compare the Numa-Aware and Numa-Blind deployment options. We deployed CONTRAST with these two modes and measured the throughput for BBB while transcoding from 4K to 1080p with different numbers of concurrent segments. Our results show that the Numa-Blind option outperforms Numa-Aware. In video streaming, videos are encoded in closed GOP format, *i.e.*, GOPs are independent of each other, for fast decodability. Thus, individual video segments are small in size and are independent of each other. The transcoding

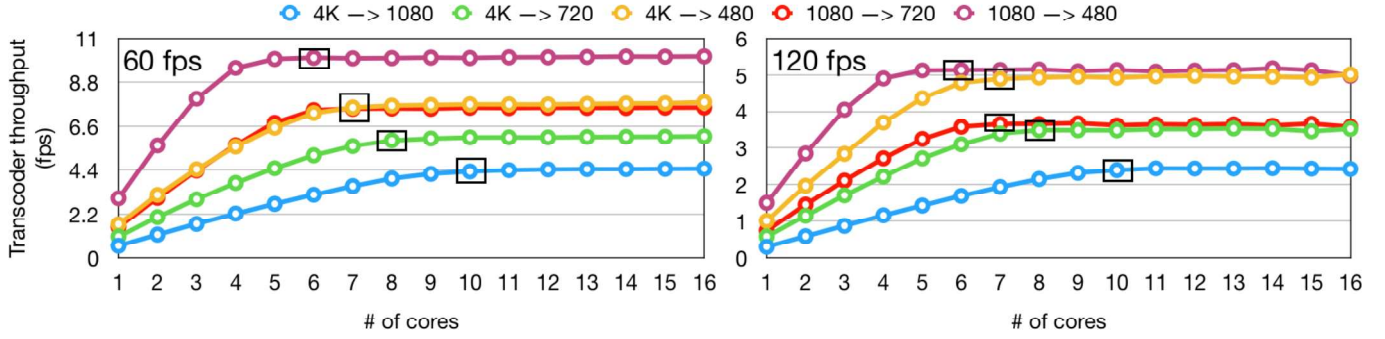


Figure 5. Profiling results for BBB with 60 fps and 120 fps frame rate

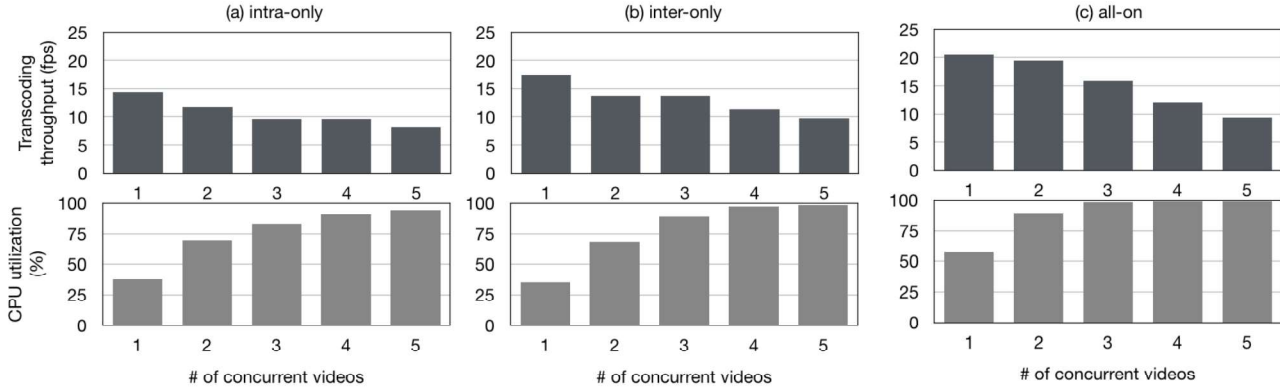


Figure 6. Comparison of different parallelism modes

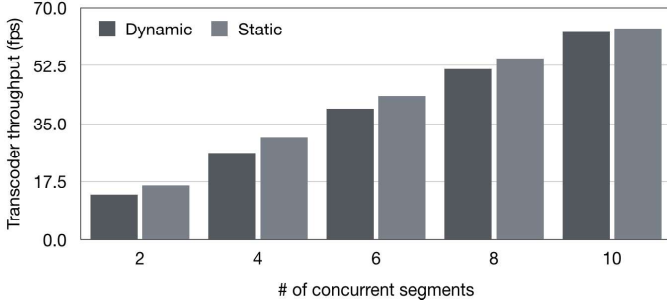


Figure 7. Comparison of dynamic and static deployment

of individual segments is done in one shot once the segment is loaded in cache and memory, hence, the process typically involves no preemption and re-caching. Given the independent and lightweight nature of per-segment transcoding, forcing the transcoding of independent segments (containers) on the same Numa node using Numa-aware does not speedup the process, but rather impacts it adversely due to the more restricted scheduling options available to the OS kernel.

C. Advantages of using Containers

This section highlights the advantages offered by containers in terms of fine-grained resource allocation. Specifically, we compare the performance of CONTRAST with a GPU system, a native system and a Spark-based system [16] that do not permit such control over the resources allocated to a transcoding operation.

In recent years Nvidia has added support for video transcoding to their data center level GPUs. They have provided libraries for common video codecs. However, the implementation and functionality of these codecs on GPUs are still

extremely limited compared to their CPU counterparts. For example, these specialized codecs do not support complete offloading of the transcoding operations on to the GPU. Portions of the transcoding still require the CPU. Despite such limitations, GPUs can offer very high transcoding speeds and consequently we compare them with CONTRAST. We use a Nvidia Tesla V100 GPU for our experiments. In the GPU deployment, the GPU is fully utilized to achieve the maximum throughput. The throughput is always the same regardless the video size. Thus, we will mostly either over provision or under provision in a live streaming setting.

We first transcode the full videos in our dataset to a 4K resolution using the GPU. We are able to transcode at most two videos concurrently. For each video, we achieved a performance of over $90fps$ as shown in Table III. The GPU did not admit more than 2 videos due to lack of GPU memory. To overcome the memory limitation, we next split up each source video into smaller $2sec$ segments to see if we can utilize the GPU better by transcoding multiple concurrent segments. Results show that the GPU is now able to handle more than 3 concurrent segments but compared to Table III the performance drops due to the segmentation. Furthermore, the performance saturates at 3 concurrent segments. The GPU delivers a maximum aggregate throughput of $80fps$ over all segments. Further inspection of GPU performance monitoring data reveals that the codec implementation is unable to utilize the GPU fully leading to the observed saturation.

From these experiments, GPUs have roughly a constant throughput and this throughput cannot be increased or decreased due to the lack of fine-grained control over GPU

Table III
GPU PERFORMANCE FOR CONCURRENT VIDEOS

Video count	Performance(fps)
1	96
2	92
3	<i>not admitted</i>

resources. Although the constant high transcoding throughput afforded by a GPU can be desirable in some applications, it is a disadvantage for interactive streaming applications. If the transcoding throughput is much higher than the continuously generated video (usually 30fps or 60 fps), the GPU is going to be idle most of the time, which is not efficient. In case of a higher frame rate video (*e.g.* 120 fps), it is difficult to scale up the GPU's performance by utilizing more of its cores since the GPU saturates at 80fps.

We next illustrate the potential cost savings due to the use of containers. We calculate the cost incurred while renting the EC2 instances we used in this study to match the peak GPU throughput of 96fps that we observe. We compare this cost with the cost of renting the same GPU as in our setup on EC2. On EC2, transcoding the BBB video using the container approach costs \$0.51 while the cost with the GPU option is \$3.60. This further illustrates the benefits of the fine-grained resource control offered by containers.

Our second comparison system is native transcoding. Instead of running containers, we run transcoding processes (*i.e.* FFMPEG) directly in the OS, and tune the number of cores to match the throughput of the container-based implementation. We first transcode each of the three videos natively using all 16 cores on one of our bare-metal machines. We then transcode the videos using containers. We adjust the number of cores assigned to a container to match the native transcoding throughput. From Fig. 8, the container approach is more resource efficient. It requires lesser number of cores and causes lower CPU utilizations than the native approach.

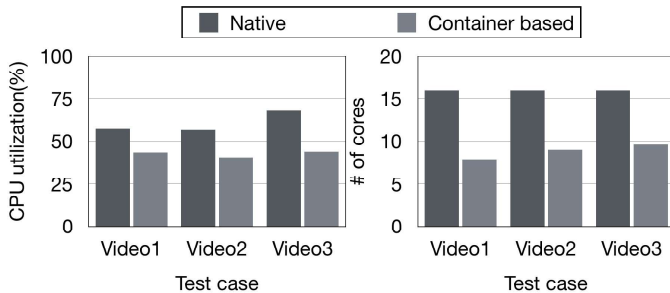


Figure 8. Comparison of native and container base transcoding.

Finally, we compare our system to Stride [16], a distributed video transcoding system based on the Apache Spark [19] big data platform. While Stride reports real-time performance, it requires the complete video to be accessible. Due to the nature of interactive streaming, CONTRAST does not impose this requirement. It only requires a few segments to be accessible. To facilitate a comparison, we modified Stride to behave as similar as possible to CONTRAST and deployed the two

systems in the same environment with the same resources. We transcoded the BBB video from 4K to 1080P using different number of concurrent segments and the target frame rate of 30 fps. In this experiment, we made sure the number of cores available to each Spark executor and containerized transcoder is identical.

The comparison results presented in Fig. 9 indicate that CONTRAST is able to achieve higher performance using the same resources. This is due to the resource efficiency and lightweight nature of containers compared to Apache Spark. Specifically, since each segment has to be submitted to Spark as a separate job to support the interactive nature of streaming, Stride is penalized by the scheduling and management overheads of Spark. It is more suited for stored videos where transcoding an entire video can be carried out as a single job.

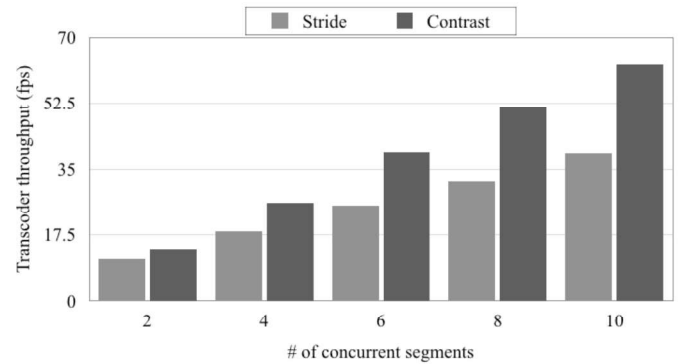


Figure 9. Performance comparison of CONTRAST and Stride

VII. CONCLUSIONS

This paper addresses the problem of transcoding interactive video sessions. Transcoding needs to be done on the fly in such systems. However, since transcoding is a compute intensive task, intelligent techniques are needed to ensure that the transcoding throughput matches the video frame rate to satisfy user QoE. We propose a container-based framework called CONTRAST that satisfies the real-time performance requirements of transcoding interactive streaming sessions in a resource efficient manner.

CONTRAST exploits our empirical finding that the degree of parallelism demanded by a transcoding operation depends solely on the source and target transcoding resolutions. Accordingly, we propose an offline profiling technique that can estimate the number of cores required for any given transcoding operation on any given hardware platform. CONTRAST launches containers right-sized with this estimate to implement the transcoding.

Future work will integrate CONTRAST with DASH-like protocols to facilitate continuous adaptation of video quality. We will also integrate CONTRAST with NFV and SDN technologies to implement transcoding capabilities at multiple points in the network, *e.g.*, the edge and the core of a network. Scaling and caching techniques to handle multiple concurrent streaming sessions and flash crowd scenarios will also be explored.

REFERENCES

- [1] “Amazon EC2.” [Online]. Available: <https://aws.amazon.com/ec2/>
- [2] “Docker.” [Online]. Available: <https://www.docker.com/>
- [3] F. Bellard, “FFMPEG,” 2015. [Online]. Available: <http://ffmpeg.org/>
- [4] K. Bilal and A. Erbad, “Edge computing for interactive media and video streaming,” in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, May 2017, pp. 68–73.
- [5] —, “Impact of Multiple Video Representations in Live Streaming: A Cost, Bandwidth, and QoE Analysis,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2017, pp. 88–94.
- [6] M. Braitmaier and D. Kyriazis, “Virtual and Augmented Reality: Improved User Experience through a Service Oriented Infrastructure,” in *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*, May 2011, pp. 40–46.
- [7] E. Casalicchio and V. Perciballi, “Measuring Docker Performance: What a Mess!!!” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE '17 Companion*. L'Aquila, Italy: ACM Press, 2017, pp. 11–16.
- [8] V. Cisco, “Cisco Visual Networking Index: Forecast and Trends, 2017–2022,” *White Paper*, 2018.
- [9] 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.
- [10] G. Gao and Y. Wen, “Morph: A fast and scalable cloud transcoding system,” in *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 2016, pp. 1160–1163.
- [11] M. V. Group, “MSU Codec Comparison 2017,” MSU Video Group, Tech. Rep., 2017. [Online]. Available: <http://www.compression.ru/video/>
- [12] T. Hoßfeld, S. Egger, R. Schatz, M. Fiedler, K. Masuch, and C. Lorentzen, “Initial delay vs. interruptions: Between the devil and the deep blue sea,” in *2012 Fourth International Workshop on Quality of Multimedia Experience*. IEEE, 2012, pp. 1–6.
- [13] P. Juluri, V. Tamarapalli, and D. Medhi, “QoE management in DASH systems using the segment aware rate adaptation algorithm,” in *Proceedings of the NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, 2016, pp. 129–136.
- [14] M. Kim, Y. Cui, S. Han, and H. Lee, “Towards efficient design and implementation of a Hadoop-based distributed video transcoding system in cloud computing environment,” *International Journal of Multimedia and Ubiquitous Engineering*, vol. 8, no. 2, pp. 213–224, 2013.
- [15] A. Kopytov, “Scriptable database and system performance benchmark: akopytov/sysbench,” May 2019, original-date: 2015-03-07T08:27:40Z. [Online]. Available: <https://github.com/akopytov/sysbench>
- [16] S. Sameti, M. Wang, and D. Krishnamurthy, “Stride: Distributed video transcoding in spark,” in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC) (IPCCC 2018)*, Orlando, USA, Nov. 2018.
- [17] —, “Container-based real-time video transcoding,” in *2019 IEEE 44th Conference on Local Computer Networks (LCN) (LCN 2019)*, Osnabrück, Germany, Oct. 2019.
- [18] C. Song, W. Shen, L. Sun, Z. Lei, and W. Xu, “Distributed video transcoding based on MapReduce,” *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, pp. 309–314, 2014.
- [19] The Apache Software Foundation, *Apache Spark*, 2014. [Online]. Available: <https://spark.apache.org>
- [20] I. Twitch Interactive, *Twitch*, 2017. [Online]. Available: <https://www.twitch.tv>
- [21] 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, 2012.
- [22] R. Wu, Y. Chen, E. Blasch, B. Liu, G. Chen, and D. Shen, “A container-based elastic cloud architecture for real-time full-motion video (FMV) target tracking,” in *2014 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, Oct. 2014, pp. 1–8.