# Cost-Efficient and Robust On-Demand Video Transcoding Using Heterogeneous Cloud Services

Xiangbo Li, Mohsen Amini Salehi, *Member, IEEE*, Magdy Bayoumi, *Fellow, IEEE*, Nian-Feng Tzeng, *Fellow, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

**Abstract**—Video streams, either in the form of Video On-Demand (VOD) or live streaming, usually have to be converted (i.e., transcoded) to match the characteristics of viewers' devices (e.g., in terms of spatial resolution or supported formats). Transcoding is a computationally expensive and time-consuming operation. Therefore, streaming service providers have to store numerous transcoded versions of a given video to serve various display devices. With the sharp increase in video streaming, however, this approach is becoming cost-prohibitive. Given the fact that viewers' access pattern to video streams follows a long tail distribution, for the video streams with low access rate, we propose to transcode them in an on-demand (i.e., lazy) manner using cloud computing services. The challenge in utilizing cloud services for on-demand video transcoding, however, is to maintain a robust QoS for viewers and cost-efficiency for streaming service providers. To address this challenge, in this paper, we present the Cloud-based Video Streaming Services (CVS2) architecture. It includes a QoS-aware scheduling component that maps transcoding tasks to the Virtual Machines (VMs) by considering the affinity of the transcoding tasks with the allocated heterogeneous VMs. To maintain robustness in the presence of varying streaming requests, the architecture includes a cost-efficient VM Provisioner component. The component provides a self-configurable cluster of heterogeneous VMs. The cluster is reconfigured dynamically to maintain the maximum affinity with the arriving workload. Simulation results obtained under diverse workload conditions demonstrate that CVS2 architecture can maintain a robust QoS for viewers while reducing the incurred cost of the streaming service provider by up to 85 percent.

**Index Terms**—Cloud services, heterogeneous VM provisioning, QoS-aware scheduling, On-demand video transcoding

## 1 Introduction

The way people watch videos has dramatically changed over the past years. From traditional TV systems, to video streaming on desktops, laptops, and smart phones through the Internet. Consumer adoption of video streaming services is rocketing. Based on the Global Internet Phenomena Report [1], video streaming currently constitutes approximately 64 percent of all U.S. Internet traffic. It is estimated that streaming traffic will increase up to 80 percent of the whole Internet traffic by 2019 [2].

Video contents, either in the form of Video On Demand (VOD) (e.g., YouTube[1] or Netflix[2]) or live-streaming (e.g., Livestream[3]), need to be converted based on the device characteristics of viewers. That is, the original video has to be converted to a supported resolution, frame rate, video codec, and network bandwidth to match the viewers' devices [3]. The conversion is termed *video transcoding* [4], which is a computationally heavy and time-consuming process [3]. One approach currently used by streaming providers for transcoding is termed *pre-transcoding*, in which several transcoded versions of a given video are stored to serve different types of devices. However, this approach requires massive storage and processing resources. In addition, recent studies (e.g., [5]) reveal that the access pattern to video streams follows a long tail distribution. That is, there is a small percentage of videos that are accessed frequently while the majority of them are accessed very infrequently. Therefore, with the explosive demand for video streaming and the large diversity of viewing devices, the *pre-transcoding* approach is inefficient.

In this research, we propose to transcode the infrequently accessed video streams in an *on-demand* (i.e., lazy) manner using computing services offered by cloud providers.

- *Xiangbo Li is with Brightcove Inc, Boston, MA 02210. E-mail: xli@brightcove.com, tzeng@cacs.louisiana.edu.*
- *Mohsen Amini Salehi is with the HPCC lab., School of Computing and Informatics, University of Louisiana at Lafayette, LA 70503. E-mail: amini@louisiana.edu.*
- *Magdy Bayoumi and Nian-Feng Tzeng are with the Center for Advanced Computer Studies, University of Louisiana at Lafayette, LA 70503. E-mail: mab@cacs.louisiana.edu.*
- *Rajkumar Buyya is with the Department of Computing and Information Systems, The University of Melbourne, Melbourne, VIC 3010, Australia. E-mail: rbuyya@unimelb.edu.au.*

*Manuscript received 17 Aug. 2016; revised 21 Sept. 2017; accepted 17 Oct. 2017. Date of publication 24 Oct. 2017; date of current version 9 Feb. 2018.*
*(Corresponding author: Xiangbo Li.)*
*Recommended for acceptance by E. Deelman.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2017.2766069*

[1] https://www.youtube.com
[2] https://www.netflix.com
[3] https://livestreams.com

The challenge for on-demand video transcoding is how to utilize cloud services to maintain a robust Quality of Service (QoS) for viewers, while incurring the minimum cost to the Streaming Service Provider (SSP).

Video stream viewers have unique QoS demands. In particular, they need to receive video streams without any delay. Such delay may occur either during streaming, due to an incomplete transcoding task by its presentation time, or at the beginning of a video stream. In this paper, we refer to the former delay as *missing presentation deadline* and the latter as the *startup delay* for a video stream. Previous studies (e.g., [5]) confirm that viewers mostly do not watch video streams to the end. However, they rank the quality of a stream provider based on the video stream's startup delay. Another reason for the importance of the startup delay is the fact that once the beginning part of a stream is processed and buffered, the provider has more time to process the rest of the video stream. Therefore, to maximize viewers' satisfaction, we define viewers' QoS demand as: *minimizing the startup delay and the presentation deadline violations*.

To minimize the network delay, transcoded streams are commonly delivered to viewers through Content Delivery Networks (CDNs) [6]. It is worth noting that, this research is not about the CDN technology. Instead, it concentrates on the computational and cost aspects of on-demand video transcoding using cloud services.

The goal of SSPs is to spend the minimum for renting cloud services, while maintaining a robust QoS for viewers. To satisfy this goal, in our earlier work [7], we investigated using homogeneous cloud Virtual Machines (VMs). One extension, we propose in this work, is to consider the fact that cloud providers offer heterogeneous types of VMs. For instance, Amazon EC2 provides General Purpose, CPU-Optimized, GPU-Optimized, Memory-Optimized, Storage-Optimized, and Dense-Storage VMs[4] with costs varying significantly. Moreover, the execution time of different transcoding operations varies on different VM types. That is, different transcoding operations have different affinities with different VM types. The challenge is how to construct a heterogeneous cluster of VMs to minimize the incurred cost of SSPs while the QoS demands of viewers are respected? More importantly, the heterogeneous VM cluster should be self-configurable. That is, based on the arriving transcoding tasks, the *number* and the *type* of VMs within the cluster should be dynamically altered to maximize the affinity with VMs and reduce the incurred cost.

Based on aforementioned definitions, the specific research questions we address in this article are:

- How can SSPs satisfy the QoS demands of viewers by minimizing both the video streaming startup delay and presentation deadline violations?
- How can SSPs minimize their incurred costs through utilizing a self-configurable heterogeneous VM cluster while maintaining a robust QoS for the viewers?

Previous works (e.g., [8], [9]) either did not consider on-demand transcoding of video streams or disregarded the specific QoS demands. Therefore, to answer these research questions, we propose the **C**loud-based **V**ideo **S**treaming **S**ervice (CVS2) architecture that enables on-demand video transcoding using cloud services. The architecture includes a scheduling component that maps transcoding tasks to cloud VMs with the goal of satisfying viewers' QoS demands. It also includes a VM Provisioner component that minimizes the incurred cost of the SSP through constructing a self-configurable heterogeneous VM cluster, while maintaining robust QoS for viewers.

In summary, the key *contributions* of this paper are as follows:

- Proposing the CVS2 architecture that enables on-demand transcoding of video streams.
- Developing a QoS-aware scheduling component within the CVS2 architecture to map the transcoding tasks to a heterogeneous VM cluster with respect to the viewers' QoS demands.
- Developing a VM Provisioner component within the CVS2 architecture that forms a self-configurable heterogeneous VM cluster to minimize the incurred cost to the SSPs while maintaining a robust QoS for viewers.
- Analyzing the behavior of the CVS2 architecture from the QoS, robustness, and cost perspectives under various workload intensities.

The rest of the paper is organized as follows. Section 2 provides a background on video streaming and transcoding. In Section 3, we present the CVS2 architecture. The scheduling and the VM provisioning policies will be discussed in Sections 4 and 5, respectively. In Section 6, we perform performance evaluations. Section 7 discusses related works in the literature, and finally Section 8 concludes the paper and provides avenues of future work.

## 2 Background

### 2.1 Definition of Robustness

*Robustness* is defined as the degree to which a system can function correctly in the presence of uncertain parameters in the system [10].

In a system for on-demand transcoding, the arrival pattern of the streaming requests is uncertain, which can significantly harm QoS and viewer satisfaction [11]. Ideally, the system has to be robust against uncertainty in the arrival pattern of the streaming requests. That is, the system has to satisfy a certain level of QoS, even in the presence of uncertain arrival of streaming requests.

### 2.2 Video Stream Structure

A Video stream, as shown in Fig. 1, consists of several sequences. Each sequence is further divided into multiple *Group Of Pictures* (GOPs) with sequence header information at the beginning. Each GOP essentially comprises a sequence of frames beginning with an `I` (intra) frame, followed by a number of `P` (predicted) frames or `B` (bi-directional predicted) frames. Each frame of a GOP contains several *slices* that consist of a number of *macroblocks* (MB) which is used for video encoding and decoding. In practice, video streams are commonly split into *GOP tasks* (simply termed GOPs in the paper) for processing that can be transcoded independently [12].

[4] https://aws.amazon.com/ec2/instance-types

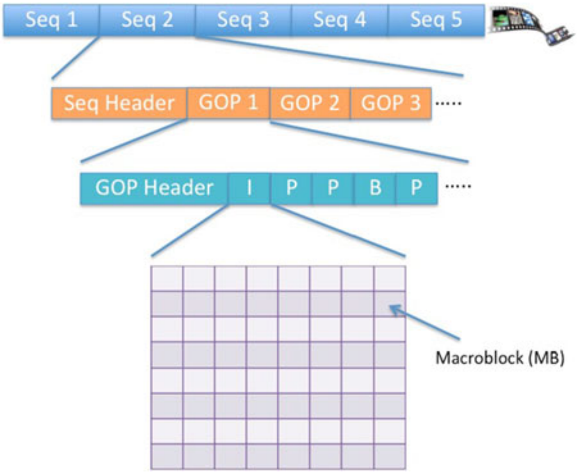


Fig. 1. The structure of a video stream. It consists of several sequences. Each sequence includes multiple GOPs. Each frame of a GOP contains several MacroBlocks.

### 2.3 Video Transcoding

A video initially is captured with a particular format, spatial resolution, frame rate, and bit rate. Then, the video is uploaded to a streaming server where it is adjusted based on the viewer's device resolution, frame rate, and video codec. These conversions are generally termed *video transcoding* [3], [4] operations and are explained as follows:

*Bit Rate Adjustment.*To produce a high quality video contents, the video is encoded with high bit rate. However, a higher bit rate also requires larger network bandwidth for video stream transmission. SSPs usually need to transcode the video stream to adjust the bit rate based on available viewer bandwidth [13].

*Spatial Resolution Reduction.*Spatial resolution indicates the encoded dimensional size of a video. However, the dimensional size does not necessarily match the screen size of the viewer's device. To avoid losing contents, macroblocks of an original video have to be removed or combined (i.e., downscaled) to produce a lower spatial resolution video [14].

*Temporal Resolution Reduction.*Temporal resolution reduction happens when the viewer's device only supports a lower frame rate, and hence, some frames have to be dropped. Due to dependency between frames, dropping frames can invalidate motion vectors (MV) for the incoming frames. Temporal resolution reduction can be achieved using methods explained in [15].

*Compression Standard (Codec) Conversion.*Video compression standards vary from MPEG2 to H.264, and to the most recent one, HEVC. MPEG2 is widely used for DVD and video broadcasting, while HD or Blu-ray videos are mostly encoded with H.264. HEVC is the latest and most efficient compression standard. Viewer devices usually support a specific codec. Thus, video streams need to be transcoded from the original codec to the one supported by the viewer's device [16].

### 2.4 Video Transcoding Using Heterogeneous VMs

Cloud providers usually offer numerous VM types. For instance, Amazon EC2 currently provides more than 40 VM types. These VM types are heterogeneous both in terms of their underlying hardware architectures and prices. In Amazon EC2, VMs are categorized in 6 groups based on their architectural configurations. In particular, these groups are: General-Purpose, CPU-Optimized, Memory-Optimized, GPU-Optimized, Storage-Optimized, and Dense-Storage.

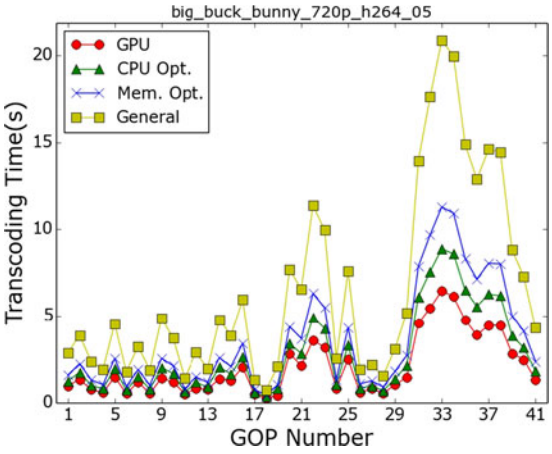


Fig. 2. Transcoding time (in seconds) of GOPs using different VM types. The horizontal axis shows the sequential order of GOP numbers in a video stream.

Our initial evaluations on transcoding the codec of a set of benchmark videos[5]: https://goo.gl/B6T5aj(explained in Section 6.1) demonstrated that transcoding GOPs have different execution times on various VM types. In particular, we executed GOPs on four VM types, and their performance results are shown in Fig. 2.[6] We did not consider any of the Storage Optimized and Dense Storage VM types in our evaluations as we observed that IO and storage are not influential factors for transcoding tasks. Due to huge diversity, we selected one VM instance that represents the characteristics within each category. More specifically, for GPU instance, CPU-Optimized, Memory-Optimized, and General-Purpose types we chose `g2.2xlarge`, `c4.xlarge`, `r3.xlarge`, and `m4.large`, respectively. The cost of the chosen instance types are illustrated in Table 1.

The vertical axis of Fig. 2 shows the transcoding time (i.e., execution time) for different GOPs of a given video stream. According to the figure, in general, GPU instances provide a lower execution time than other VM instance types. However, for some of the GOPs, the performance difference of GPU with other VM instances is negligible, while its cost is remarkably higher (see Table 1). The experiment indicates that an SSP can utilize heterogeneous VM types to minimize its incurred cost while satisfying viewers' QoS demands.

## 3 CVS2: Cloud-Based Video Streaming Service Architecture

### 3.1 Overview

The CVS2 architecture aims to deal with a received request for streaming a video format that is not available in the repository (i.e., it is not pre-transcoded). An overview of the architecture is presented in Fig. 3. It shows the sequence of actions taken place to transcode a video stream in an on-demand manner. The dashed lines in this figure will be investigated in our future studies.

[5] the workload trace of the benchmark videos are available from

[6] Fig. 2 shows the result for one of the benchmark videos. We used big buck bunny 720p in the benchmark for this experiment. However, results for other experiments confirm the same observations.

TABLE 1
Cost of Different VM types in Amazon EC2

| VM Type | GPU | CPU Opt. | Mem. Opt. | General |
|---|---|---|---|---|
| | (g2.xlarge) | (c4.xlarge) | (r3.xlarge) | (m4.large) |
| Hourly Cost ($) | 0.65 | 0.20 | 0.33 | 0.15 |

CVS2 architecture includes eight main components, namely *Video Splitter*, *Admission Control*, *Time Estimator*, *Task (i.e., GOP) Scheduler*, *Heterogeneous Transcoding VMs*, *VM Provisioner*, *Video Merger*, and *Caching*. These components are explained in the next few sections.

### 3.2 Video Splitter

The Video Splitter splits the video stream into several GOPs that can be transcoded independently. Each generated GOP is identified uniquely in form of $G_{ij}$, where $i$ is the video stream id and $j$ is the GOP number within the video stream.

Each GOP is treated as a task with an individual deadline. The deadline of a GOP is the presentation time of the first frame in that GOP. In the case of VOD, if a GOP misses its deadline, it still has to complete its transcoding. We have made the source code for Video Splitter publicly[7] available.

### 3.3 Admission Control

The Admission Control component includes policies that regulate GOP dispatching to the scheduling queue. In fact, the Video Splitter generates GOPs for all requested video streams. Then, the admission control policies determine the priority (i.e., urgency) of the GOPs and dispatches them accordingly to the scheduling queue. The admission control policies act based on the inputs it receives from Video Splitter and Video Merger.

The way Admission Control prioritizes a GOP is based-on the GOP sequence number in a video stream. Details of how to prioritize GOP tasks is explained in Section 4.2

### 3.4 Transcoding Virtual Machines (VMs)

VMs are allocated from the cloud provider to transcode GOP tasks. As discussed in Section 2.4, cloud providers offer VMs with diverse architectural configurations. Although GOPs can be processed on all VM types, their execution times vary. In fact, the execution time of a GOP on a particular VM type can depend on factors such as the size of data it processes or the type of transcoding operations it performs.

Each VM is assigned a local queue where the required data for GOPs are preloaded before execution. The scheduler maps GOPs to VMs until the local queue gets full.

### 3.5 Execution Time Estimator

The role of the Time Estimator component is to estimate the execution time of GOP tasks. Such estimation of execution times helps the Scheduler and VM Provisioner components to function efficiently.

In VOD streaming, a video usually has been streamed multiple times. Therefore, the transcoding execution time for each $G_{ij}$ can be estimated from the historic execution information of $G_{ij}$ [17].

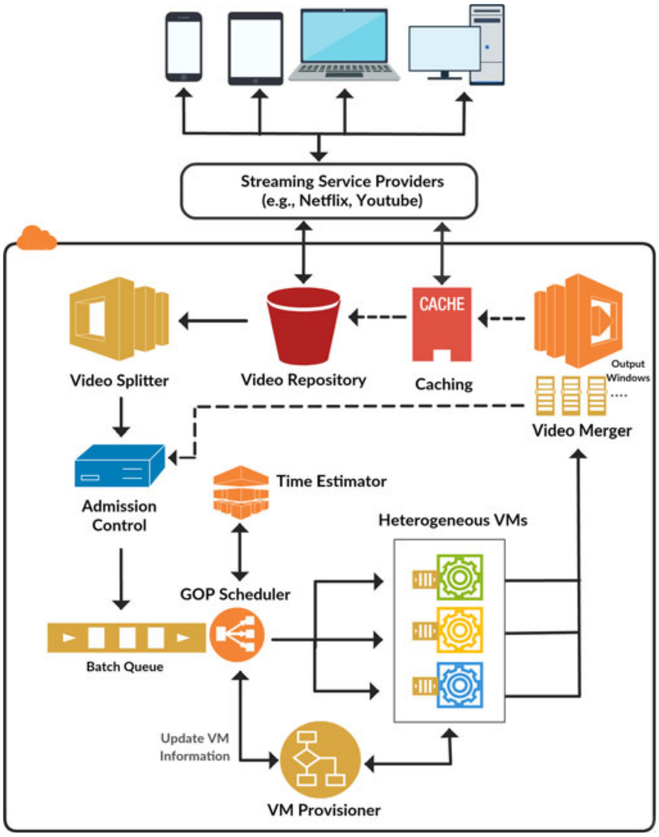


Fig. 3. An overview of the Cloud-based Video Streaming Service (CVS2) architecture.

As we consider the case of heterogeneous transcoding VMs, each GOP has a different execution time on each VM type. Therefore, the Time Estimator stores the execution time estimations within Estimated Time to Completion (ETC) matrices [10]. An entry of the ETC matrix expresses the execution time of a given GOP $G_{ij}$ on a given VM type $m$.

We note that, even in transcoding the same GOP $G_{ij}$ on the same type of VM, there is some randomness (i.e., uncertainty) in the transcoding execution time. That is, the same VM type does not necessarily provide identical performance for executing the same GOP at different times [18]. This variance is attributed to the fact that the same VM type can be potentially allocated on different physical machines on the cloud. It can also be attributed to other neighboring VMs that coexist with the VM on the same physical host in the cloud datacenter. For instance, if the neighboring VMs have a lot of memory access, then, there will be a contention to access the memory and the performance of the VM will be different from the situation that there is no such a neighboring VM. Therefore, to capture randomness that exists in the GOP execution time, the mean execution time and its standard deviation of the historic execution time for $G_{ij}$ is stored in the corresponding entry of the ETC matrix.

### 3.6 Transcoding (GOP) Task Scheduler

The GOP task scheduler (briefly called transcoding scheduler) is responsible for mapping GOPs to a set of heterogeneous VMs. Considering the heterogeneity in performance and cost of different VM types, the scheduler's goal is to

[7] The source code for GOP task generation is available here: https://github.com/lxb200709/videotranscoding_gop

map GOP tasks to VMs with the minimum incurred cost while satisfying the QoS demands of the viewers.

GOPs of different video streams are interleaved within the scheduling queue. In addition, the scheduler has no prior knowledge about the arrival pattern of the GOPs to the system. Details of the scheduling method are presented in Section 4.

### 3.7 VM Provisioner

The VM Provisioner component monitors the operation of transcoding VMs in the CVS2 architecture and dynamically reconfigures the VM cluster with two goals: (A) minimizing the incurred cost to the stream provider; (B) maintaining a robust QoS for viewers. For that purpose, the VM Provisioner includes provisioning policies that are in charge of *allocating* and *deallocating* VM(s) from the cloud based on the streaming demand type and rate.

VM provisioning policies generally have to determine *when* and *how many* VMs need to be provisioned (known as elasticity [7]). For a heterogeneous VM cluster, the policy also has to determine *which type* of VM needs to be provisioned.

The VM provisioning policies are executed periodically and also in an event-based fashion to verify whether or not the allocated VMs are sufficient to meet the QoS demands. Once the provisioning policy updates the set of allocated VMs, it informs the scheduler about the latest configuration of the VM cluster. Details of the VM provisioning policies are discussed in Section 5.

### 3.8 Video Merger

GOPs are transcoded on different VMs independently. Thus, latter GOPs in a video stream may be completed before the earlier ones in a stream. The role of Video Merger is to rebuild the sequence of GOPs in the right order. To build the transcoded stream, Video Merger maintains an output window for each video stream.

Video Merger is in contact with the Admission Control component. In the event that a GOP is delayed (e.g., due to failure) the Video Merger asks the Admission Control for resubmission of the GOP. Upon receiving a resubmission request, Admission Control fetches the requested GOP from Splitter and resubmits it to the Scheduler with a high priority.

Video Merger requests for resubmission of a GOP after a certain time elapsed and it does not need to search for the missed GOP to see if it has failed or not.

### 3.9 Caching

To avoid redundant transcoding of the trending videos, the CVS2 architecture provides a caching policy to decide whether a transcoded video should be stored or not. If the video is barely requested by viewers, there is no need to store (i.e., cache) the transcoded version. Such videos are transcoded in an on-demand manner upon viewers' request. We will explore more details of the caching policy in a future research.

Considering the proposed architecture, in the next two sections, we elaborate on the methods developed for the Transcoding Task Scheduler and VM Provisioner components.

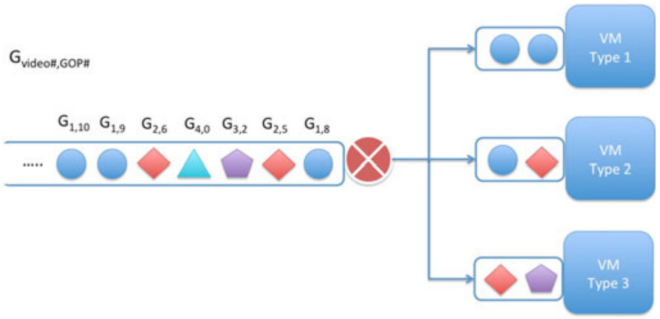


Fig. 4. QoS-aware transcoding scheduler that functions based on the utility value of the GOPs.

## 4 QoS-Aware Transcoding (GOP) Task Scheduler

### 4.1 Overview

Details of the GOP task scheduler are shown in Fig. 4. According to the scheduler, GOPs of the requested video streams are batched in a queue upon arrival to be mapped to VMs by the scheduling method. To avoid any execution delay, the required data for GOPs are fetched in the local queue of the VMs, before the GOP transcoding started. Previous studies [10] show that the local queue size should be short. Accordingly, we consider the local queue size to be 2 in all VMs. We assume that the GOP tasks in the local queue are scheduled in the *first come first serve* (FCFS) fashion. Once a free slot appears in a VM local queue, the scheduling method is notified to map a GOP task from those in the batch queue to the free slot. We assume that GOP scheduling is non-preemptive and non-multi-tasking.

Recall that the scheduler goal is to satisfy the QoS demands of viewers by minimizing the average deadline miss rate and the average startup delay of the video streams. The scheduling method maps the GOP tasks to a heterogeneous cluster of VMs where GOPs have different execution times on different VM types. In such a system, optimal mapping of GOP tasks to heterogeneous VMs is an NP-complete problem [19]. Thus, development of mapping heuristics to find near-optimal solutions forms a large body of research [10], [20].

In the rest of this section, we explain the details of how the scheduling component within the CVS2 architecture satisfies the QoS demands. Also, for further clarity, all the symbols used in this paper are listed in Table 2, in Appendix A Section, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2017.2766069.

### 4.2 Utility-based GOP Task Prioritization

One approach to minimize the average startup delay of video streams is to consider a separate dedicated queue for the startup GOPs of the streams [7]. Such a queue can only prioritize a constant number of GOPs at the beginning of the streams, with the rest of the GOPs treated as normal priority. In practice, however, the priority of GOPs should be decreased gradually as the video stream moves forward.

To implement the gradual prioritization of GOPs in a video stream, we define a *utility function* that operates on a video stream and assigns *utility values* to each GOP.

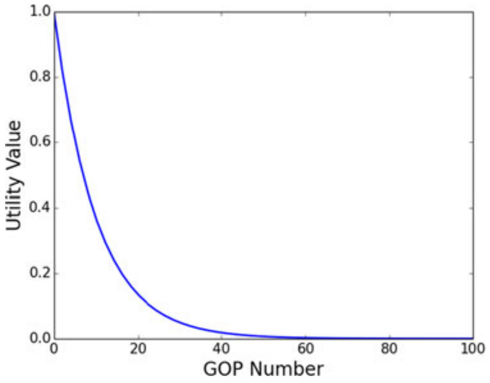


Fig. 5. Utility values of different GOP tasks to indicate their processing priority within a video stream.

Equation (1) shows the utility function the admission control policy uses for assigning utility values. In Equation (1), $c$ is a constant and $i$ is the order number of GOP in the video stream. The value of $c$ determines the slope of the utility function curve. That means, using this parameter we can adjust the importance of the startup GOPs in a video stream. Higher values for $c$ create a sharp slope in the curve that implies prioritizing few GOPs in the beginning of the video stream with a high utility value and low utility values for the rest of GOPs in the video stream. Our initial experiments showed that $c = 0.1$ provides a reasonable slope in Equation (1).That is, it assigns a high utility value to the GOPs in the beginning of the stream and then the utility value gradually decreases for GOPs positioned later in the stream.

$$U_i = \left(\frac{1}{e}\right)^{c \cdot i} \tag{1}$$

The utility values assigned to a given video stream are depicted in Fig. 5. In this figure, the horizontal axis is the GOP number and the vertical axis is the utility value. As we can see, the utility function assigns higher utility values (i.e., higher priority) to earlier GOPs in the stream. The utility value drops for the latter GOPs in the stream.

We would like to note that, although we used Equation (1) to assign utility values to GOP tasks, our proposed method is general and its operation is not dependent on this particular utility function. In fact, our proposed methods can operate under any utility function as long as it assures that the first part of the video is prioritized more than the rest of it.

### 4.3 Estimating Task Completion Time on Heterogeneous VMs

For each GOP $j$ from video stream $i$, denoted $G_{ij}$, the arrival time and the deadline (denoted $\delta_{ij}$) are available. It is worth noting that the GOP deadline is relative to the beginning of the video stream. Therefore, to obtain the absolute deadline for $G_{ij}$ (denoted $\Delta_{ij}$) the relative deadline must be added to the presentation start time of the video stream (denoted $\psi_i$). That is, $\Delta_{ij} = \delta_{ij} + \psi_i$.

Recall that the estimated execution time for $G_{ij}$ on VM type $m$ is available through the ETC matrix (see Section 3.5). To capture randomness in the estimated execution time of GOPs, let $\tau_{ij}^m$ be the worst-case transcoding time estimation. That is, in the scheduling, we consider $\tau_{ij}^m$ as the sum of mean historic execution times of $G_{ij}$ and its standard deviation on $VM_m$.

Our scheduling method also needs to estimate the tasks' *completion times* to be able to efficiently map them to VMs. To estimate the completion time of an arriving GOP task $G_n$ on $VM_m$, we add up the estimated remaining execution time of the currently executing GOP in $VM_m$ with the estimated execution time of all tasks ahead of $G_n$ in the local queue of $VM_m$. Finally, we add the estimated execution time of $G_n$ (i.e., $\tau_n^m$). Recall that each GOP task has a different execution time on different VM types that can be obtained from the ETC matrix (see Section 3.5). Let $t_r$ denote the remaining estimated execution time of the currently executing task on $VM_m$, and let $t_c$ be the current time. Then, we can estimate the *task completion time* of $G_n$ on $VM_m$ (denoted $\varphi_n^m$) as follows:

$$\varphi_n^m = t_c + t_r + \sum_{p=1}^{N} \tau_p^m + \tau_n^m \tag{2}$$

where $\tau_p^m$ denotes the worst case estimated execution time of any task waiting ahead of $G_n$ in local queue of $VM_m$ and $N$ is the number of waiting tasks in local queue of $VM_m$.

### 4.4 Mapping Heuristics

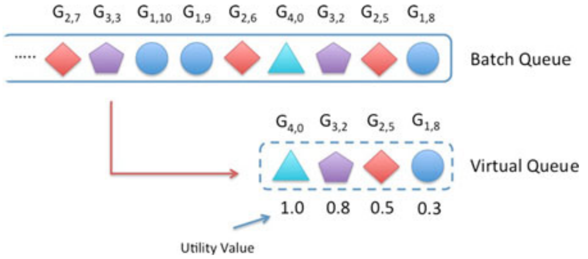
Mapping heuristics are responsible to map tasks from the batch queue to machine queues (see Fig. 4).Regardless of their implementation details, mapping heuristics for heterogeneous computing systems have a general mechanism that operates in two main phases [21]. In Phase 1, for all tasks in the batch queue, the machine (i.e., VM) that provides the minimum expected completion time is determined. The output of this phase can be considered as pairs of tasks with the machines that provide the minimum expected completion time for them. Then, in Phase 2, from the set of task-machine pairs identified in Phase 1, the mapping heuristic selects the pair that maximizes its performance objective. This process is repeated until either all tasks in the batch queue are assigned or there is no free slot left in machine queues.

Based on the explained mechanism, MinCompletion-MinCompletion (MM) [22], [23], [24], [25], MinCompletion-SoonestDeadline (MSD) [10], [26], and MinCompletion-MaxUrgency (MMU) [10], [26] mapping heuristics are defined as follows:

*MinCompletion-MinCompletion (MM).*In Phase 1, the heuristic finds the machine (i.e., VM) that provides the minimum expected completion time for the GOP task. In Phase 2, the heuristic selects the pair that has the minimum completion time from all the task-machine pairs generated in the Phase 1. Once the selected task is mapped to the selected machine, it is removed from the batch queue.

*MinCompletion-SoonestDeadline (MSD).*In Phase 1, for each task in the batch queue, the heuristic finds the VM that provides the minimum expected completion time. In Phase 2, from the list of task-machine pairs found in the Phase 1, MSD assigns the task that has the soonest deadline.

*MinCompletion-MaxUrgency (MMU).*In Phase 1 of MMU, for each task in the batch queue, the heuristic finds the VM that provides the minimum expected completion time. In Phase 2, from the list of task-machine pairs found in the Phase 1, MMU assigns the task whose task urgency is the greatest (i.e., has the shortest slack).



$G_{2,7}$ $G_{3,3}$ $G_{1,10}$ $G_{1,9}$ $G_{2,6}$ $G_{4,0}$ $G_{3,2}$ $G_{2,5}$ $G_{1,8}$
.....
Batch Queue
$G_{4,0}$ $G_{3,2}$ $G_{2,5}$ $G_{1,8}$
Virtual Queue
1.0 0.8 0.5 0.3
Utility Value

Fig. 6. Virtual Queue to hold GOPs with the highest utility values from different video streams. GOPs in Virtual Queue are ready for mapping to VMs.

Although these mapping heuristics are extensively employed in heterogeneous computing systems, none of them consider the task precedence based on the utility value as discussed in Section 4.2.

### 4.4.1 Utility-Based Mapping Heuristics

Recall that each GOP is assigned a utility value that shows its precedence. Therefore, in the *first phase* of our proposed scheduling method, as shown in Fig. 6, the GOPs with the highest utility values are selected and put into a virtual queue. The rest of the scheduling method is applied on the virtual queue rather than the whole batch queue. Given the large number of GOPs in the batch queue, making use of the virtual queue reduces the scheduling overhead.

In the *second phase*, similar to the heuristics introduced in Section 4.4, task-VM pairs are formed based on the VM that provides the minimum expected completion time for each GOP in the priority queue. Then, in the *third phase*, the mapping decision is made by combining a performance objective (e.g., SoonestDeadline) and the utility values of the GOP tasks. For combining, we prioritize the GOP with the highest utility value from the pairings of a VM, if and only if it does not violate the deadline of the task selected based on the performance objective.

To clarify further, we explain the third phase using an example. Let GOP tasks $G_a$ and $G_b$ denote pairs for $VM_m$. Also, let SoonestDeadline be the performance objective. Assume that $G_a$ has a sooner deadline, whereas $G_b$ has a higher utility value. In this case, $G_b$ can be assigned to $VM_m$, if and only if it does not violate the deadline of $G_a$. To assure that assigning $G_b$ does not cause a violation of the deadline of $G_a$, we assume that $G_b$ has already been assigned to $VM_m$ and run the mapping heuristic again to see if $G_a$ can still meet its deadline or not.

Based on the way the third phase of our proposed mapping heuristic functions, we can have 3 variations, namely Utility-based MinCompletion-MinCompletion (MMUT), Utility-based MinCompletion-SoonestDeadline (MSDUT), and Utility-based MinCompletion-MaximumUrgency (MMUUT).

# 5 Self-Configurable Heterogeneous VM Provisioner

## 5.1 Overview

The goal of the VM Provisioner component is to maintain a robust QoS while minimizing the incurred cost to the stream provider. To that end, the component includes VM provisioning policies that make decisions for *allocating* and *deallocating* VMs from cloud.

To achieve the QoS robustness, the SSP needs to define the acceptable QoS boundaries. Therefore, the SSP provides an upper bound threshold for the deadline miss rate of GOPs that can be tolerated, denoted $\beta$. Similarly, it provides a lower bound threshold for the deadline miss rate, denoted $\alpha$, that enables the provisioning policies to reduce the incurred cost of the stream provider through deallocating VM(s).

The strategy of the VM provisioning to maintain QoS robustness is to manage the VM allocation/deallocation so that the deadline miss rate at any given time $t$, denoted $\gamma_t$, remains between $\alpha$ and $\beta$. That is, at any given time $t$, we should have $\alpha \leq \gamma_t \leq \beta$.

The VM Provisioner component follows the *scale up early and scale down slowly* principle. That is, VM(s) are allocated from the cloud as soon as a provisioning decision is made. However, as the stream provider has already paid for the current charging cycle of the allocated VMs, the deallocation decisions are not practiced until the end of the current charging cycle.

In general, any cloud-based VM provisioning policy needs to deal with two main questions:

1) *When* to provision VMs?
2) *How many* VMs to provision?

The self-configurable VM provisioning, however, introduces a third question to the VM provisioning policies:

3) *What type* of VM(s) to provision?

In the next sections, we first provide a method to determine the suitability of VM types for GOP tasks, then we introduce two provisioning policies, namely periodic and remedial, that work together to answer the three aforementioned questions.

## 5.2 Identifying Suitability of VM Types for GOP Tasks

Recall that each GOP task has different execution times on different VM types (see Section 2). In general, GPU provides a shorter execution time compared with other VM types. However, for some GOPs, the execution time on GPU is close to other VM types while its cost is significantly higher (see Table 1). Therefore, we need a measure to determine the suitability of a VM type for a GOP based on the two factors.

For a given GOP task, we define *suitability*, denoted $S_i$, as a measure to quantify the appropriateness of a VM type $i$ for executing the GOP task both in terms of performance and cost. We calculate the suitability measure for a task based on Equation (3). The measure establishes a trade-off between the performance ($T_i$) and the cost ($C_i$) for a given GOP on VM type $i$.

$$S_i = k \cdot T_i + (1 - k) \cdot C_i \tag{3}$$

The value of $k$, in Equation ( 3), is determined by the CVS2 user (i.e., video stream provider) and represents her preference between performance and cost of VM type $i$. The value of $T_i$ is defined based on Equation (4).

$$T_i = \frac{t_{max} - t_i}{t_{max} - t_{min}} \tag{4}$$

where $t_i$ is the GOP execution time on VM type $i$ (obtained from the ETC matrix). Also, $t_{max}$ and $t_{min}$ are the maximum and minimum GOP execution times across all VM types, respectively. Nominator of this equation determines the execution time improvement provided by VM type $i$ for the GOP. Denominator of this equation ensures that the value of $T_i$ remains in [0,1] space.

In Equation (3), the value of $C_i$ is determined according to Equation (5).

$$C_i = \frac{c_{max} - c_i}{c_{max} - c_{min}} \tag{5}$$

where $c_i$ is the cost of transcoding the same GOP on VM type $i$. Also, $c_{max}$ and $c_{min}$ are the maximum and minimum GOP transcoding costs across all VMs, respectively. The rationale of Equation ( 5) is similar to that of Equation ( 4). Nominator of the equation determines the cost improvement resulted from VM type $i$ to transcode the GOP and denominator ensures the value of $C_i$ remains in [0,1].

Based on Equation (3), for a given GOP task, we define the *GOP type* based on the type of VM that provides the highest suitability value. Later, the VM provisioning policies will utilize the concept of GOP type in their provisioning decisions.

## 5.3 Periodic VM Provisioning Policy

This VM provisioning policy occurs periodically (we term it *provisioning event*) to make VM allocation or deallocation decisions. The policy includes two methods, namely *Allocation* and *Deallocation*.

### 5.3.1 Allocation Method

Algorithm 1 provides a pseudo-code for the VM allocation method. The method is triggered *when* the deadline miss rate ($\gamma_t$) goes beyond the upper bound threshold $\beta$ (line 2 in the Algorithm). The value of $\beta$ is determined by the video streaming service provider (i.e., CVS2 user) and represents how much the provider can tolerate QoS violation in favor of cost-efficiency.

**Algorithm 1.** Pseudo-Code for the VM Allocation Method

```
Input:
    $\beta$: upper bound threshold for deadline miss rate
    $r$: streaming request arrival rate
Output:
    $n$: list of number of VMs of each type to be allocated.
1:  $\gamma_t \leftarrow$ current deadline miss rate
2:  if $\gamma_t \geq \beta$ then
3:      for each VM type $i$ do
4:          $\sigma_i \leftarrow$ deadline miss rate for each GOP type $i$
5:          $\phi_i \leftarrow$ ratio of each GOP type $i$ in the batch queue
6:          Calculate the demand ($\omega_i$) for each VM type $i$
7:          $\rho_i \leftarrow$ minimum utilization in VMs of type $i$
8:          if $\omega_i \geq \omega_{th}$ and $\rho_i \geq \rho_{th}$ then
9:              $n_i \leftarrow \lfloor \frac{r \cdot \omega_i}{\beta} \rfloor$
10:             Allocate $n_i$ VM type $i$
11:         end if
12:     end for
13: end if
```

To determine *what type* of VM(s) to be allocated, we need to understand the demand for different VM types. Such demand can be understood from the concept of GOP type, introduced in Section 5.2. In fact, the number of GOP tasks from different types can guide us to the types of VMs that are required. More specifically, we can identify the type of required VMs based on two factors: (A) the proportion of deadline miss rate for each GOP type, denoted $\sigma_i$, and (B) the proportion of GOPs of each type waiting for execution in the batch queue, denoted $\phi_i$. In fact, factor (A) indicates the current QoS violation status of the system, whereas factor (B) indicates the QoS violation status of the system in the near future.

Based on these factors, we define the *demand* for each VM type $i$, denoted $\omega_i$, according to Equation (6). The constant factor $0 \leq k \leq 1$, in this equation, determines the weight assigned to the current deadline miss rate status and to the future status of the system.

For implementation, we experimentally realized that the value of $k$ should be determined in a way that GOPs waiting in the batch queue (i.e., $\phi_i$) are assigned a higher weight, rather than the current QoS violation of each GOP type (i.e., $\sigma_i$). The reason is that, the GOP tasks in the batch queue represent the QoS violation the system will encounter in a near future which is more important than the QoS violation the system currently is encountering. Hence, we considered $k = 0.3$ (thus, $1 - k = 0.7$) in Equation ( 6). Based on this justification, we believe that in a system with a different workload scenario than those we considered in our evaluations, the value of $k$ should remain the same.

$$\omega_i = k \cdot \sigma_i + (1 - k) \cdot \phi_i \tag{6}$$

If the demand for VM type $i$ is greater than the allocation threshold ($\omega_{th}$ in line 8), and also the utilization of corresponding VM type ($\rho_i$) is greater than the utilization threshold ($\rho_{th}$), then the policy decides to allocate from VM type $i$.

Once we determine the type of VMs that needs to be allocated, the last question to be answered is *how many* VMs of each type to be allocated (lines 8 - 11 in the Algorithm). The number of allocations of each VM type depends on how far is the deadline miss rate of GOP type $i$ is from $\beta$. For that purpose, we use the ratio of $\omega_i/\beta$ to determine the number of VM(s) of type $i$ that has to be allocated (line 9). The number of VM(s) allocated also depends on the arrival rate of GOP tasks to the system. Therefore, the GOP arrival rate, denoted $r$, is also considered in line 9 of Algorithm 1.

### 5.3.2 Deallocation Method

The VM deallocation method functions are based on the lower bound threshold ($\alpha$). That is, it is triggered *when* the deadline miss rate ($\gamma_t$) is less than $\alpha$. Once the deallocation method is executed, it terminates at most one VM. The reason is that, if the VM deallocation decision is practiced aggressively, it can cause loss of processing power and results in QoS violation in the system. Therefore, the only question in this part is *which* VM should be deallocated.

In the first glance, it seems that the deallocation method can simply choose the VM with the lowest utilization for deallocation. However, this is not the case when we are dealing with a heterogeneous VM cluster. The utilizations of the VMs are subject to the degree of heterogeneity in the VM cluster. For instance, when the VM cluster is in a mostly homogeneous configuration, the task scheduler has no tendency to a particular VM type. This causes all VMs in the cluster to have a similar and high utilization. Hence, if the deallocation method functions just based on the utilization, it cannot terminate VM(s) in a homogeneous cluster, even if the deadline miss rate is low.

The challenge is how to identify the degree of heterogeneity in a VM cluster. To cope with this challenge, we need to quantify the VM cluster heterogeneity. Then, we can apply the appropriate deallocation method accordingly.

We define *degree of heterogeneity*, denoted $\eta$, as a quantity that explains the VM diversity (i.e., heterogeneity) that exists within the current configuration of the VM cluster. We utilize the Shannon Wiener equitability [27] function to quantify the degree of heterogeneity within our VM cluster. The function works based on the Shannon Wiener Diversity Index that is represented in Equation (7).

$$H = -\sum_{i=1}^{N} p_i \cdot \ln p_i \tag{7}$$

where, $N$ is the number of VM types, $p_i$ is the ratio of VM type $i$ of the total number of VMs. Then, the degree of heterogeneity is defined as follows:

$$\eta = H/H_{max} \tag{8}$$

Higher values of $\eta$ indicates a higher degree of heterogeneity in a cluster and vice versa. Once we know the degree of heterogeneity in a VM cluster, we can build the deallocation method accordingly. Algorithm 2 provides the pseudo-code proposed for the VM deallocation method. The method is triggered *when* the deadline miss rate ($\gamma_t$) becomes less than the lower bound threshold $\alpha$, which is defined by the CVS2 user and represents how much the system can tolerate deadline miss rate in favor of cost-efficiency.

**Algorithm 2.** Pseudo-Code for the VM Deallocation Method

```
Input:
    $\alpha$: lower bound threshold for deadline miss rate
 1: $\gamma_t \leftarrow$ current deadline miss rate
 2: if $\gamma_t \leq \alpha$ then
 3:     calculate the utilization of each VM in the cluster
 4:     find VM(s) with the lowest utilization
 5:     resolve ties by choosing the least powerful VM(s)
 6:     $VM_j \leftarrow$ resolve ties by selecting the VM with the minimum remaining time to its charging cycle
 7:     $\eta \leftarrow$ calculate the degree of heterogeneity
 8:     if $\eta \geq \eta_{th}$ and $\rho_j \geq \rho_{th}$ then
 9:         No deallocation
10:     else
11:         Deallocate $VM_j$
12:     end if
13: end if
```

The deallocation method is carried out in 4 main steps. In the first step, the VM(s) with the lowest utilization are chosen (lines 3–4 in Algorithm 2). In the second step, ties are broken by selecting the least powerful VM (line 5). If more than one VM remains, in the third step (line 6), ties are broken based on the VM with the minimum remaining time to its charging cycle.

For a VM cluster that tends to a heterogeneous configuration (i.e., $\eta \geq \eta_{th}$), the policy deallocates the selected VM (termed $VM_j$ in the algorithm) if its utilization is less than the VM utilization threshold (i.e., $\rho_j < \rho_{th}$). The value of $\eta_{th}$ determines the boundary between homogeneous and heterogeneous configurations in a VM cluster. We experimentally realized that $\eta_{th} = 0.4$ can discriminate homogeneous configurations from heterogeneous ones. The value of $\rho_{th}$ is determined by the CVS2 user based on its cost and performance trade-off. In contrast, in a VM cluster that tends to a homogeneous configuration, even if the utilization is high, the policy can deallocate $VM_j$ based on the deadline miss rate (lines 8–12).

It is worth noting that the deallocation method is also executed at the end of the charging cycle of the current VMs to deallocate VMs marked for deallocation. The reason for enacting VM termination at the end of the VM charging cycle is that the VM has already been paid for the whole charging cycle. Therefore, there is no benefit in terminating it before its charging cycle, even though it is recommended for deallocation. To implement this and to assure that no GOP task is left incomplete, the scheduler keeps track of each VM's remaining time to its charging cycle and the completion time of the tasks assigned to that VM. If a VM is marked for deallocation, before scheduler maps a new GOP task to it, the scheduler estimates the completion time of GOPs assigned to that VM, in addition to the completion time of the new GOP task. If the completion times are larger than the time remains to the VM's charging cycle, the GOP tasks are rescheduled on other VMs. Otherwise, the scheduler keeps sending GOP tasks to the VM, even though it is marked for deallocation.

### 5.4 Remedial VM Provisioning Policy

The periodic VM provision policy cannot cover request arrivals to the batch queue that occur in the interval of two provisioning events.

To cope with the shortage of the periodic policy, we propose a lightweight remedial provisioning policy that can improve the overall performance of the VM Provisioner component. By injecting this policy into the intervals of the periodic provisioning policy, we can perform the periodic policy less frequently.

In fact, the remedial provisioning policy provides a quick prediction of the system based on the state of the virtual queue. Recall that the Virtual Queue includes the distinction of streaming requests waiting for transcoding in the batch queue. Hence, the length of the Virtual Queue implies the intensity of streaming requests waiting for processing. Such long batch queue increases the chance of a QoS violation in the near future. Thus, our lightweight remedial policy only checks the size of the Virtual Queue (denoted $Q_s$). Then, it uses Equation (9) to decide for the number of VMs that should be allocated.

$$n = \left\lfloor \frac{Q_s}{\theta \cdot \beta} \right\rfloor \tag{9}$$

where $n$ is the *number* of VM(s) that should be allocated; $Q_s$ is the size of the Virtual Queue. $\theta$ is a constant factor that determines the aggressiveness of the VM allocation in the policy. That is, lower values of $\theta$ leads to allocating more VMs and vice versa. In the implementation, we considered $\theta = 10$. In the remedial policy, we allocate a VM type that, in general, provides a high performance per cost ratio (in the experiments, we used `c4.xlarge`).

Experiment results indicate that the remedial provisioning policy does not incur any extra cost to the stream service provider. Nonetheless, it increases the robustness of the QoS by reducing the average deadline miss rate and average startup delay (see Section 6.5). To verify the performance of the proposed methods, in the next section, we evaluate them in different configurations and under various workload conditions.

## 6 Performance Evaluation

### 6.1 Experimental Setup

We used CloudSim [28], a discrete event simulator, to model our system and evaluate performance of the scheduling methods and VM provisioning policies. To create a diversity of video streaming requests, we uniformly selected videos over the range of [10, 600] seconds from a set of benchmark videos. We made the benchmarking videos publicly available for reproducibility purposes.[8] We modeled our system based on the characteristics and cost of VM types in Amazon EC2. We considered `g2.2xlarge`, `c4.xlarge`, `r3.xlarge`, and `m4.large` in our experiments. The VMs represent the characteristics of various VM types offered by Amazon cloud and form a heterogeneous VM cluster.

To simulate a realistic video transcoding scenario, using `FFmpeg`,[9] we performed four different transcoding operations (namely codec conversion, resolution reduction, bit rate adjustment, and frame rate reduction) for each of the benchmarking videos. Then, the execution time of each transcoding operation was obtained by executing them on the different VM types.

To capture the randomness in the execution time of GOPs on cloud VMs, we transcoded each GOP 30 times and modeled the transcoding execution times of GOPs based on the Normal distribution.[10]

To study the performance of the system comprehensively, we evaluated the system under various workload intensities. For that purpose, we varied the arrival rate of the video streaming requests from 100 to 1000 within the same period of time. The inter-arrival times of the requested videos are generated based on the Normal distribution, where the mean of inter-arrival time is based on the time divided by the number of requests and standard deviation is the mean divided by 3. All experiments of this section were run 30 times, and the mean and the 95 percent of the confidence interval of the results are reported for each experiment. In all the experiments, we considered the values of $\alpha$ and $\beta$ equal to 0.05 and 0.1, respectively. That is, we consider that the SSP chose to keep the deadline miss rate between 5 percent to 10 percent. Any deadline miss beyond 10 percent is considered as a *QoS violation*. The QoS boundary is shown in the form of a horizontal line in the experiment results.

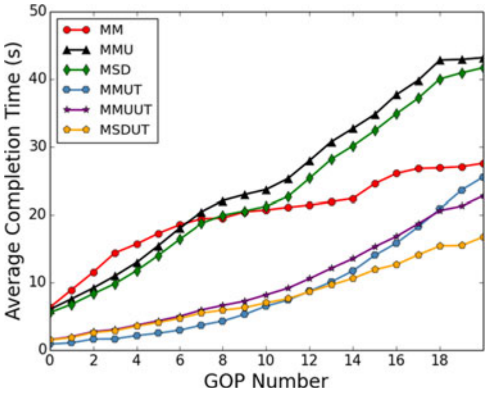


Fig. 7. Average completion time of early GOPs under different scheduling methods. The horizontal axis shows the GOP numbers in the video stream and the vertical axis shows the average completion time of GOPs. We used 1000 GOP tasks and the VM provisioning policies are applied.

### 6.2 Average Completion Time of Early GOP Tasks

The goal of using utility-based mapping heuristics is to prioritize GOPs with high utility (i.e., earlier GOPs in the stream) for reducing their completion time. Although this factor is extended in next experiments through evaluating the average startup delay. We conduct the experiment to further evaluate how this goal is satisfied when our utility-based scheduling methods with different mapping heuristics are applied.

In Fig. 7, the horizontal axis is the GOP number of the first 20 GOPs in the benchmark video streams and the vertical axis is the average completion time of the GOPs in seconds. For this experiment, we have used 1000 GOP tasks and VM provisioning policies are in place.

Fig. 7 demonstrates that, in general, the utility-based heuristics provide a significantly lower average completion time. Among traditional heuristics, MM performs the best. This is because MM prioritizes the GOPs with short execution times, which results in faster processing in the system. We also observed that MSDUT performs better in compare with other utility-based heuristics, specifically for GOP numbers more than 15. This is because the dynamic VM provisioning policy works based on the tasks deadline miss rate. Since MSDUT favors tasks with short deadlines, many GOPs miss their deadlines as the system becomes busy. Therefore, it allocates more VMs that, in turn, reduces the average completion time of the GOPs.

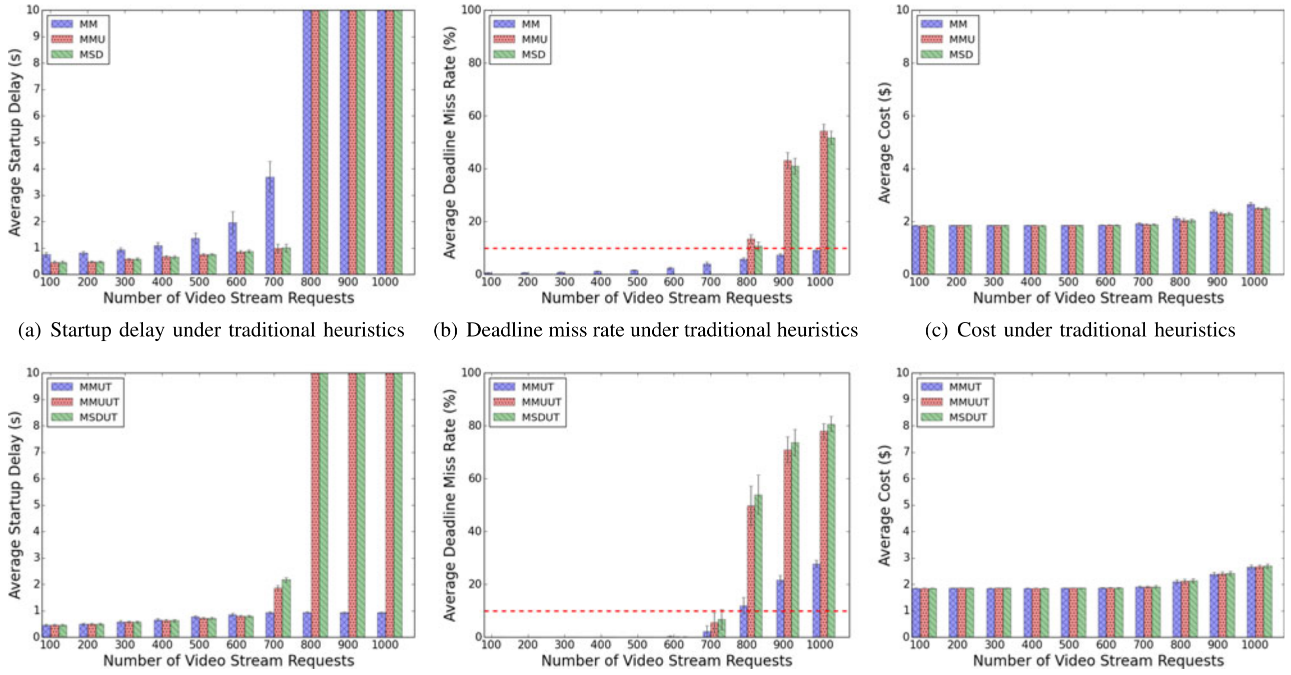
### 6.3 Impact of Utility-Based Mapping Heuristics

To evaluate the impact of utility-based mapping heuristics on QoS and cost, we compare them with the traditional mapping heuristics in two scenarios: (1) VM provisioning performed in the static way (Section 6.3.1) and (2) under the VM provisioning policies (Section 6.3.2). To construct a static heterogeneous cluster, we allocate three VMs of each type.

[8] The videos can be downloaded from: https://goo.gl/TE5iJ5

[9] https://ffmpeg.org

[10] The generated workload traces are available publicly from: https://goo.gl/B6T5aj



(a) Startup delay under traditional heuristics (b) Deadline miss rate under traditional heuristics (c) Cost under traditional heuristics

(d) Startup delay under utility-based heuristics (e) Deadline miss rate under utility-based heuristics (f) Cost under utility-based heuristics

Fig. 8. The results under utility-based mapping heuristics against those under traditional mapping heuristics when the number of video requests varies. Subfigures (a), (b), and (c), respectively, show the average startup delay, deadline miss rate, and the incurred cost under traditional mapping heuristics, while (d), (e), and (f) show the same factors under utility-based mapping heuristics are applied. The horizontal dashed line denotes the acceptable QoS boundary ($\beta$).

### 6.3.1 Static Heterogeneous VM Cluster

Fig. 8 compares the results of utility-based mapping heuristics with the traditional batch heuristics under a static heterogeneous VM cluster. For traditional mapping heuristics, Fig. 8a and 8b show that MM provides a significantly lower average deadline miss rate (by up to 40 percent) than MSD and MMU, in particular when the system is more oversubscribed (i.e., overloaded). However, MSD and MMU provide a lower average startup delay than MM. This is because both MSD and MMU function based on the deadline and the deadline of the startup GOPs is low since they are prioritized.

In Fig. 8e, we observe that MMUT provides a significantly better average deadline miss rate (around 50 percent when there are 1000 video requests) in comparison with MSDUT and MMUUT. More importantly, we can see, in Fig. 8d, that MMUT provides a low and stable startup delay in comparison with other heuristics even when the system is oversubscribed. This is because prioritizing shorter tasks in MMUT produces a lower average deadline miss rate which, in return, benefits the startup GOPs to be processed.

We should note that although MMUT provides a lower start up delay, it yields a higher deadline miss rate than the traditional MM (see Fig. 9). This is because the utility-based mapping heuristics prioritize GOPs with higher utility values (i.e., higher priority) to reduce the start up delay. This causes a higher deadline miss rate particularly when we use static resource allocation. As we will explain in the next section, utility-based mapping heuristics, in particular MMUT, significantly outperform traditional mapping heuristics, when accompanied with dynamic resource provisioning.

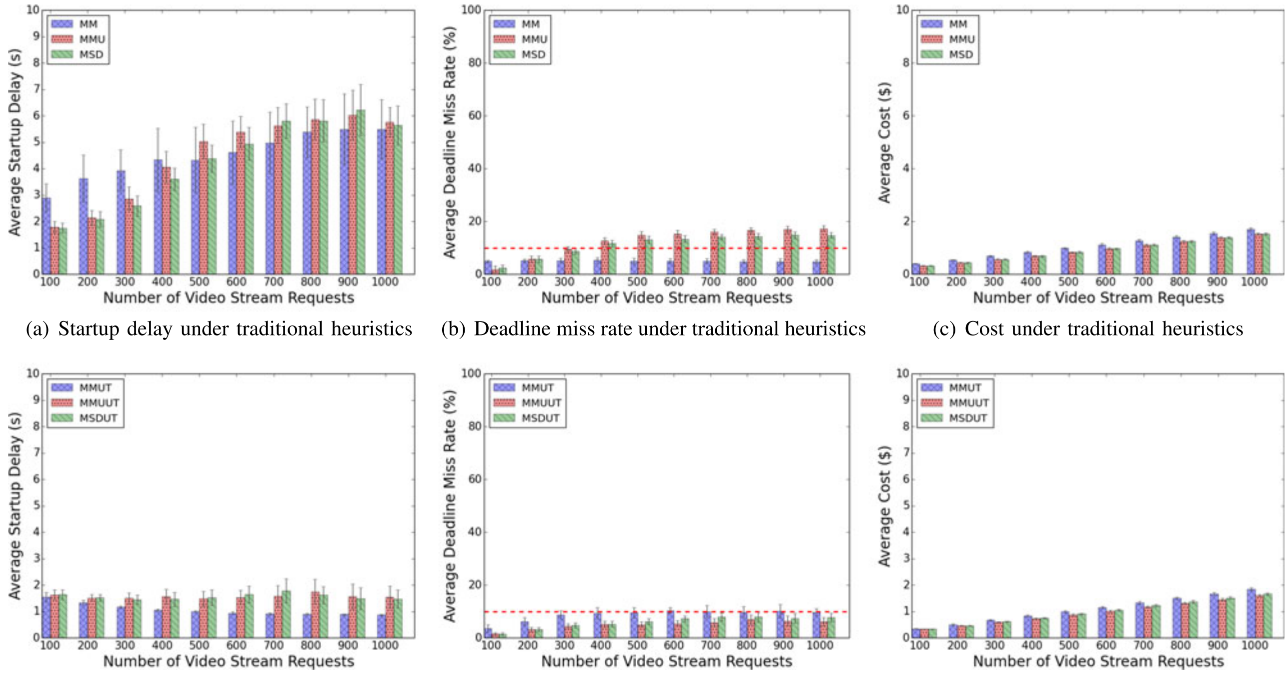
We do not observe any major cost difference for more intensive workloads. This is because in the static cluster, the workload can be handled within the same time period. When the system is oversubscribed, there is a minor increase in cost, as seen in Figs. 8c and 8f. This is because it takes a longer time to finish the processing of the tasks in those cases.

### 6.3.2 Dynamic Heterogeneous VM Cluster

Fig. 9c demonstrates that, regardless of the mapping heuristic, the dynamic VM provisioning policy significantly reduces the incurred cost (up to 80 percent when the system is not oversubscribed) in comparison to the static heterogeneous VM cluster. The incurred cost increases as the VM provisioning policy needs to allocate additional VMs to maintain QoS robustness for more video streaming requests.

In Fig. 9a, we can observe that the average startup delay increases for traditional mapping heuristics. However, it is more stable in comparison with Fig. 8a with static heterogeneous VMs. This is because the VM provisioning policy adapts the VM provisioning to the workload intensity to meet the QoS demands of the stream viewers.

Figs. 9d, 9e, and 9f demonstrate the robustness resulted from applying the utility-based mapping heuristics together with the VM provisioning policies. That is, with the increase of the workload, the system all together produces a low and stable average startup delay and average deadline miss rate without incurring extra cost to the stream provider. In particular, we observe the average deadline miss rates of MMUUT and MSDUT have dramatically decreased. Normally, MMUUT and MSDUT lead to higher average deadline miss rates than MMUT. However, with the dynamic VM provisioning policies, the high deadline miss rates of MMUUT and MSDUT trigger the VM provisioning policies to allocate more VMs that, in turn, reduce the deadline miss



(a) Startup delay under traditional heuristics (b) Deadline miss rate under traditional heuristics (c) Cost under traditional heuristics

(d) Startup delay under utility-based heuristics (e) Deadline miss rate under utility-based heuristics (f) Cost under utility-based heuristics

Fig. 9. The results under utility-base mapping heuristics against those under traditional mapping heuristics when dynamic provisioning policies are applied. The $X$-axis indicates the number of streaming requests, and Subfigures (a), (b), and (c) show the average startup delay, deadline miss rate, and the incurred cost, respectively, under traditional mapping heuristics, while (d), (e), and (f) show the same factors under utility-based mapping heuristics. The horizontal dashed line indicates the acceptable QoS boundary ($\beta$).

rate. Nonetheless, the deadline miss rate of MMUT is not sufficiently high enough to trigger the allocation method.

Further evaluations and comparisons against previous works are discussed in Appendix B, available in the online supplemental material.

#### 6.3.3 Discussion

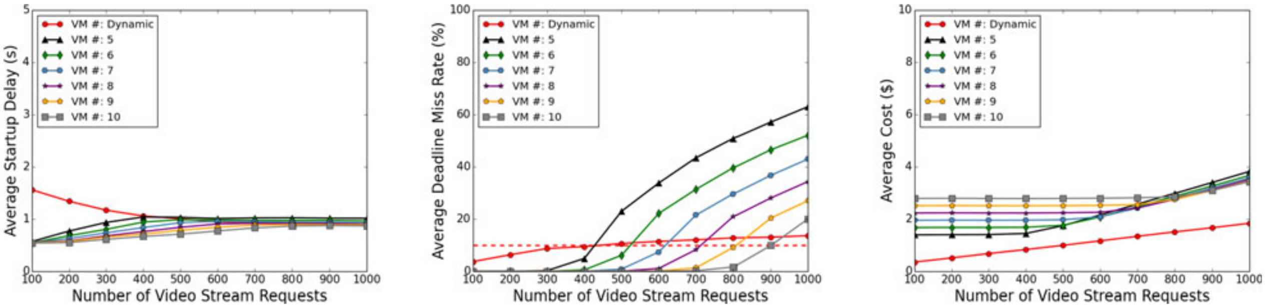
We can summarize our findings about the proposed mapping heuristics (discussed in Sections 6.3.1 and 6.3.2) as follows:

1) In both static and dynamic heterogeneous VM provisioning: MMUT provides the lowest and the most stable average startup delay in compare with all other mapping heuristics.
2) In both static and dynamic heterogeneous VM provisioning: The three proposed mapping heuristics incur approximately the same cost to the stream provider.
3) In static heterogeneous VM provisioning: MMUT results in a lower average deadline miss rate, in compare with MMUUT and MSDUT.
4) In dynamic heterogeneous VM provisioning: MMUUT and MSDUT outperform MMUT in terms of average deadline miss rate. Typically, MMUUT and MSDUT result in a higher deadline miss rate (as shown in Fig. 8e, when a static VM provisioning is used). The reason for the opposite behavior of MMUUT and MSDUT, in dynamic VM provisioning, is that their higher deadline miss rate triggers allocating more VMs, hence, their deadline miss rate is decreased. It is worth noting that, although MMUT results in a higher deadline miss rate, it is still below the threshold provided by the video stream provider (see Fig. 9e).

### 6.4 The Impact of VM Provisioning Policies

To further investigate the performance of the proposed VM provisioning policies, we compare it against the case in which a static homogeneous VM cluster is deployed. For evaluation, we vary the number of streaming requests in the system from 100 to 1000. In this experiment, we choose MMUT as the mapping heuristic. The reason for choosing MMUT is that, in general, it performs better than other heuristics both in static and dynamic VM provisioning. Albeit, MMUT does not outperform other heuristics in terms of deadline miss rate when dynamic VM provisioning is used (see Fig. 9e). However, even in that case, it can still keep the deadline miss rate below the QoS threshold provided by the video stream provider For the static clusters, as it is shown in Fig. 10, we evaluate clusters with 5 to 10 VMs. In all of them we utilized GPU VM type (`g2.2xlarge`). We observed that the average startup delay, and the average deadline miss rate are too high when fewer VMs are allocated. Therefore, we do not include them in the graphs. We would like to note that we also used other VM types to compare against dynamic VM provisioning. However, their performances were even worse than the GPU type.

In Fig. 10a, we can see that as the number of video requests increases, the average startup delay in all static policies grows. However, the dynamic VM provisioning policy provides a low and stable average startup delay. When the system is not oversubscribed (i.e., number of stream requests less than 400), the dynamic provisioning policies provide a slightly higher startup delay than the static policy. The reason is that when the deadline miss rate is low, the VM provisioning policies allocate fewer VMs to reduce the incurred cost. Hence, new GOP tasks have to wait for transcoding. However, in the static policy, specifically with a



(a) Comparison of average startup delay (b) Comparison of average deadline miss rate (c) Comparison of average cost

Fig. 10. Performance comparison under static and dynamic VM provisioning policies. Subfigure (a) illustrates the average startup delay, (b) shows the average deadline miss rate, and (c) demonstrates the incurred cost to the streaming provider under dynamic and static provisioning policies, with MMUT applied as the mapping heuristic.

large number of VMs, GOPs in the startup queue can be transcoded quickly, reducing the average startup delay.

Fig. 10b illustrates that the VM provisioning policies lead to low and stable average deadline miss rate in comparison with the static ones. In the static configuration, as the number of video requests increases, the average deadline miss rate grows dramatically. However, we notice that the average deadline miss rate with the dynamic VM provisioning policies remains stable, even when the system becomes oversubscribed. We can conclude from the experiment that the proposed VM Provisioner component in the CVS2 enables the system to tolerate workload oversubscription. That is, it makes the system robust against the fluctuations in the arrival workload.

In addition to low and stable average startup delay and average deadline miss rate, Fig. 10c shows that the dynamic VM provisioning policies reduce the incurred cost by up to 85 percent when the system is not oversubscribed. Even when the system is oversubscribed (i.e., with more than 500 streaming requests in the system) the dynamic VM provisioning policies reduced the cost to around 50 percent. In fact, when the streaming request rate is low, VMs are under-utilized; however, in the static VM cluster, the streaming service provider still has to pay for them. In contrast, with the dynamic VM provisioning, the system deallocates idle VMs when the deadline miss rate is low, which reduces the incurred cost significantly. As the number of streaming requests increases, more VMs of the appropriate types are created, and hence, the incurred cost of the dynamic VM provisioning policies approaches that of the static one. We can conclude that, from the cost perspective, our proposed VM provisioning policies are more efficient, particularly when the system is lightly loaded.

### 6.5 Impact of the Remedial VM Provisioning Policy

To evaluate the efficacy of the remedial provisioning policy, we conduct an experiment on the dynamic VM provisioning policy in two scenarios: (A) when the VM Provisioner component uses both the periodic and remedial polices and (B) when only the periodic provisioning policy is in place. We measure QoS in terms of average Deadline Miss Rate (DMR), average startup delay, and the incurred cost when the number of streaming requests varies in the system (along the $X$-axis in Fig. 11). In this experiment we assume that the MMUT mapping heuristic is utilized.

As illustrated in Fig. 11, when the system is not oversubscribed (i.e., fewer than 500 streaming requests), the difference between the two scenarios is negligible. This is because when streaming requests arrived between two provisioning events are not excessive, the VMs allocated by the periodic VM provisioning policy are sufficient to keep the QoS robust.

Alternatively, when the system is oversubscribed, the number of streaming requests that arrive between two provisioning events is high and affects the prediction of the provisioning policy. Under this circumstance, as depicted in Fig. 11, relying only on the periodic provisioning policy leads to a high deadline miss rate. Nonetheless, when the remedial VM provisioning policy is utilized even with the system is oversubscribed, the deadline miss rate remains stable. In addition, as it is shown in the last sub-figure of Fig. 11, the remedial VM provisioning policy comes without incurring any extra cost to the stream provider.

## 7 Related Work

Techniques, architectures, and challenges of video transcoding have been investigated by Ahmad et al. [3] and Vetro et al. [4]. Cloud-based video transcoding for VOD has been studied in [29], [30]. However, they all investigated the case of offline transcoding (i.e., pre-transcoding). A taxonomy of the researches undertaken on cloud-based video transcoding and the position of our contribution with respect to them is illustrated in Fig. 12.

Jokhio et al. [31] present a computation and storage trade-off strategy for cost-efficient video transcoding in the

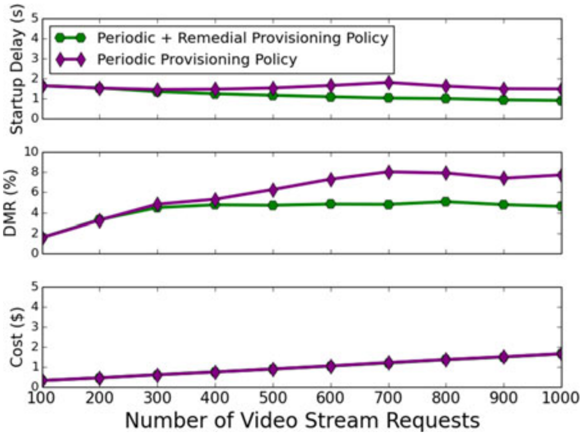


Fig. 11. Impact of the remedial VM provisioning policy on the startup delay, deadline miss rate (DMR) and the incurred cost.

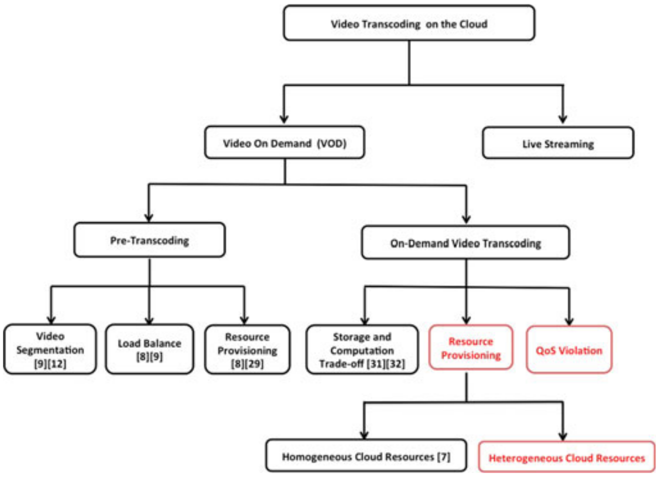

Video Transcoding on the Cloud
Video On Demand (VOD)
Live Streaming
Pre-Transcoding
On-Demand Video Transcoding
Video Segmentation [9][12]
Load Balance [8][9]
Resource Provisioning [8][29]
Storage and Computation Trade-off [31][32]
Resource Provisioning
QoS Violation
Homogeneous Cloud Resources [7]
Heterogeneous Cloud Resources

Fig. 12. A taxonomy of video transcoding using cloud. Red blocks position the contributions of this work.

cloud. The trade-off is based on the computation cost versus the storage cost of the video streams. They determine how long a video should be stored or how frequently it should be re-transcoded from a given source video. Zhao et al. [32] take the popularity, computation cost, and storage cost of each version of a video stream into account to determine versions of a video stream that should be stored or transcoded. The earlier studies demonstrate that it is possible to transcode infrequently accessed videos streams in an on-demand manner [33]. However, they do not explore the possible ways to carry out the on-demand transcoding efficiently by utilizing appropriate scheduling methods and VM provisioning policies.

In systems with dynamical task arrival, task scheduling can be performed either in an *Immediate* or a *Batch* mode [20]. In the former, the tasks are mapped to processing machines as soon as they arrive to the scheduler, whereas in the latter, few tasks are collected in a batch queue and are scheduled at the same time. Amini Salehi et al. [10] have compared these scheduling types in heterogeneous computing systems and concluded that the batch-mode significantly outperforms the immediate-mode. The reason is that, in the batch-mode, tasks can be shuffled and they do not have to be assigned in the order they arrived. Accordingly, we consider batch-mode mapping in the scheduling component of the CVS2 architecture. It is noteworthy that the current batch-mode scheduling heuristics (e.g., see those in [20]) and even those in the immediate-mode cannot fulfill the QoS requirements of on-demand video transcoding applications, mainly in terms of the startup delay.

To consider the startup delay, in [7], a startup queue was considered to prioritize the first few GOPs in video streams. Alternatively, in this paper, we improve the startup queue model by assigning a utility value to each GOP. To minimize the startup delay, the earlier GOPs in a video stream are assigned higher utility values.

Ashraf et al. [8] propose a stream-based admission control and scheduling approach using a two-step prediction model to foresee the upcoming streams' rejection rate through predicting the waiting time at each machine. Later, a job scheduling method is utilized to drop some video segments to prevent video transcoding jitters. However, they do not consider minimizing the startup delay of video stream using a heterogeneous cluster of VMs.

Previous works on cloud-based VM provisioning for video transcoding (e.g., [9], [30]) mostly consider the case of off-line transcoding. Thus, their focuses are mainly on reducing makespan (i.e., total transcoding times) and the incurred costs.

Netflix adopts the *scale up early, scale down slowly* principle for its VM provisioning [34] on Amazon EC2. It periodically checks the utilization of its allocated VMs. The allocated VMs are scaled up by 10 percent, if their utilization is greater than 60 percent for 5 minutes. They are also scaled down by 10 percent, if the VMs utilizations is less than 30 percent for 20 minutes. Lorido et al. [34] categorize current auto-scaling techniques into five main families: static threshold-based rules, control theory, reinforcement learning, queuing theory, and time series analysis. Then, they utilize the classification to carry out a literature review of proposals for auto-scaling in the cloud.

In our earlier works [7], [35], a QoS-aware VM provisioning policy was proposed for on-demand video transcoding. Nonetheless, the policy did not consider heterogeneous types of VMs offered by cloud providers. They just consider one type of VM (i.e., a homogeneous cluster of VMs) and try to minimize the incurred cost to the stream provider. Given the affinity between different transcoding types and VM types, VM provisioning policies are required to allocate and deallocate from heterogeneous VM types to minimize the incurred cost. This will enable the creation of a dynamically-formed VM cluster that changes its configurations based on the arriving transcoding requests. The current work is different from [7] in several other ways too. We provide a method to quantify heterogeneity of a VM cluster and use it in deallocation policy of the VM cluster. We provide a method to quantify the suitability of each VM type for various transcoding operations. We develop new scheduling heuristics that are QoS-aware and are tailored for heterogeneous computing systems. We also provide a utility function that prioritizes GOPs in a video stream based on their position in the stream.

## 8 Conclusions and Future Work

In this paper, we proposed the CVS2 streaming engine for on-demand video transcoding. In particular, we developed the Task Scheduler and VM Provisioner components of CVS2. The components are aware of the viewers' QoS demands and aim to maintain QoS robustness while minimizing the incurred cost to the SSP. The components take advantage of the heterogeneous VMs, offered by the cloud providers with diverse prices. The Scheduler minimizes the startup delay and the deadline violations of the streams. The VM Provisioner is cost-aware in allocating/deallocating heterogeneous VMs. Experiment results demonstrate that proposed scheduling reduces the average startup delay and the deadline miss rate. In addition, heterogeneous VM provisioning reduces the incurred cost by up to 85 percent, particularly, when the system is not oversubscribed. The VM provisioning is robust against uncertainties in the arrival of streaming requests, without incurring any extra cost to the provider.

The CVS2 architecture is useful for SSPs to utilize cloud services and offer on-demand transcoding of video streams with a low cost. In future, we will extend the admission control to be failure-aware. We will also consider multiple cloud scenarios for faster video delivery.

## Acknowledgments

This research was supported by the Louisiana Board of Regents under grant number LEQSF(2016-19)-RD-A-25. This is a substantially extended version of a paper presented at the IEEE/ACM International on Conference on Cluster, Cloud and Grid Computing (CCGrid '16) [7].

## References

[1] G. I. P. Report. (2016, Oct.). [Online]. Available: https://www.sandvine.com/trends/global-internet-phenomena/

[2] C. V. N. Index, "Forecast and methodology, 2014-2019," May 2015.

[3] I. Ahmad, X. Wei, Y. Sun, and Y.-Q. Zhang, "Video transcoding: An overview of various techniques and research issues," *IEEE Trans. Multimedia*, vol. 7, no. 5, pp. 793–804, Oct. 2005.

[4] A. Vetro, C. Christopoulos, and H. Sun, "Video transcoding architectures and techniques: An overview," *IEEE Mag. Signal Process.*, vol. 20, no. 2, pp. 18–29, Mar. 2003.

[5] X. Cheng, J. Liu, and C. Dale, "Understanding the characteristics of internet short video sharing: A youtube-based measurement study," *IEEE Trans. Multimedia*, vol. 15, no. 5, pp. 1184–1194, Aug. 2013.

[6] R. Buyya, M. Pathan, and A. Vakali, *Content Delivery Networks*, vol. 9. Berlin, Germany: Springer, 2008.

[7] X. Li, M. A. Salehi, M. Bayoumi, and R. Buyya, "CVSS: A cost-efficient and QoS-aware video streaming using cloud services," in *Proc. 16th IEEE/ACM Int. Conf. Cluster Cloud Grid Comput.*, May 2016, pp. 106–115.

[8] A. Ashraf, F. Jokhio, T. Deneke, S. Lafond, I. Porres, and J. Lilius, "Stream-based admission control and scheduling for video transcoding in cloud computing," in *Proc. 13th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, May 2013, pp. 482–489.

[9] 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," *Int. J. Multimedia Ubiquitous Eng.*, vol. 8, no. 2, pp. 213–224, Mar. 2013.

[10] M. A. Salehi, et al., "Stochastic-based robust dynamic resource allocation for independent tasks in a heterogeneous computing system," *J. Parallel Distrib. Comput.*, vol. 97, pp. 96–111, Jun. 2016.

[11] M. Maurer, I. Brandic, and R. Sakellariou, "Adaptive resource configuration for cloud infrastructure management," *Future Generation Comput. Syst. J.*, vol. 29, no. 2, pp. 472–487, Feb. 2013.

[12] F. Jokhio, T. Deneke, S. Lafond, and J. Lilius, "Analysis of video segmentation for spatial resolution reduction video transcoding," in *Proc. IEEE Int. Symp. Intell. Signal Process. and Commun. Syst.*, Dec. 2011, pp. 1–6.

[13] O. Werner, "Requantization for transcoding of mpeg-2 intraframes," *IEEE Trans. Image Process.*, vol. 8, pp. 179–191, Feb. 1999.

[14] N. Bjork and C. Christopoulos, "Transcoder architectures for video coding," *IEEE Trans. Consum. Electron.*, vol. 44, no. 1, pp. 88–98, Feb. 1998.

[15] S. Goel, Y. Ismail, and M. Bayoumi, "High-speed motion estimation architecture for real-time video transmission," *Comput. J.*, vol. 55, no. 1, pp. 35–46, Apr. 2012.

[16] T. Shanableh, E. Peixoto, and E. Izquierdo, "MPEG-2 to HEVC video transcoding with content-based modeling," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 23, pp. 1191–1196, Jul. 2013.

[17] A. M. Al-Qawasmeh, A. A. Maciejewski, and H. J. Siegel, "Characterizing heterogeneous computing environments using singular value decomposition," in *Proc. 24th IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, Apr. 2010, pp. 1–9.

[18] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *Proc. 1st Int. Conf. Cloud Comput.*, Oct. 2009, pp. 115–131.

[19] P. B. Bhat, C. S. Raghavendra, and V. K. Prasanna, "Efficient collective communication in distributed heterogeneous systems," *J. Parallel Distrib. Comput.*, vol. 63, no. 3, pp. 251–263, 2003.

[20] B. Khemka, et al., "Utility functions and resource management in an oversubscribed heterogeneous computing environment," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2394–2407, Aug. 2015.

[21] J. Smith, A. A. Maciejewski, and H. J. Siegel, "Maximizing stochastic robustness of static resource allocations in a periodic sensor driven cluster," *Future Generation Comput. Syst. J. (FGCS)*, vol. 33, pp. 1–10, Apr. 2014.

[22] J. L. L. Simarro, R. M. Vozmediano, F. Desprez, and J. R. Cornabas, "Image transfer and storage cost aware brokering strategies for multiple clouds," in *Proc. 7th IEEE Int. Conf. Cloud Comput.*, Jun. 2014, pp. 737–744.

[23] L. D. Briceno, et al., "Heuristics for robust resource allocation of satellite weather data processing on a heterogeneous parallel system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, pp. 1780–1787, Jan. 2011.

[24] V. Shestak, J. Smith, A. A. Maciejewski, and H. J. Siegel, "Stochastic robustness metric and its use for static resource allocations," *J. Parallel Distrib. Comput.*, vol. 68, no. 8, pp. 1157–1173, Aug. 2008.

[25] J.-K. Kim et al., "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment," *J. Parallel Distrib. Comput.*, vol. 67, no. 2, pp. 154–169, Feb. 2007.

[26] J. E. Smith, J. Apodaca, A. A. Maciejewski, and H. J. Siegel, "Batch mode stochastic-based robust dynamic resource allocation in a heterogeneous computing system." in *Proc. 16th Int. Conf. Parallel Distrib. Process. Techn. Appl.*, pp. 263–269, Jul. 2010.

[27] I. F. Spellerberg and P. J. Fedor, "A tribute to claude shannon (1916-2001) and a plea for more rigorous use of species richness, species diversity and the 'shannon–wiener'index," *Global Ecology Biogeography*, vol. 12, no. 3, pp. 177–179, May 2003.

[28] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw.: Practice Experience*, vol. 41, pp. 23–50, Aug. 2011.

[29] F. Jokhio, A. Ashraf, S. Lafond, I. Porres, and J. Lilius, "Prediction-based dynamic resource allocation for video transcoding in cloud computing," in *Proc. 21st IEEE Int. Conf. Parallel Distrib. Netw.-Based Process.*, Feb. 2013, pp. 254–261.

[30] S. Lin, X. Zhang, Q. Yu, H. Qi, and S. Ma, "Parallelizing video transcoding with load balancing on cloud computing," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2013, pp. 2864–2867.

[31] F. Jokhio, A. Ashraf, S. Lafond, and J. Lilius, "A computation and storage trade-off strategy for cost-efficient video transcoding in the cloud," in *Proc. 39th EUROMICRO Conf. Softw. Eng. Adv. Appl.*, Sep. 2013, pp. 365–372.

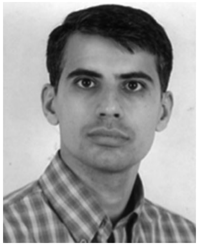
[32] H. Zhao, Q. Zheng, W. Zhang, B. Du, and Y. Chen, "A version-aware computation and storage trade-off strategy for multi-version VoD systems in the cloud," in *Proc. 20th IEEE Symp. Comput. Commun.*, Jul. 2015, pp. 943–948.

[33] K. Keahey and M. Parashar, "Enabling on-demand science via cloud computing," *IEEE Trans. Cloud Comput.*, vol. 1, no. 1, pp. 21–27, Aug. 2014.

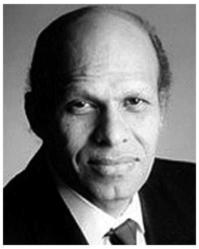
[34] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, Mar. 2014.

[35] X. Li, M. A. Salehi, and M. Bayoumi, "High perform on-demand video transcoding using cloud services," in *Proc. 16th IEEE/ACM Int. Conf. Cluster Cloud Grid Comput.*, ser. CCGrid'16, May 2016, vol. 16, pp. 600–603.

**Xiangbo Li** received the PhD degree in computer engineering from the University of Louisiana at Lafayette, in 2016. He is currently working as video engineer with Brightcove Inc., a cloud based online video platform company. He is an expert in cloud-based video encoding, transcoding, and packaging.



**Mohsen Amini Salehi** received the PhD in computing and information systems from Melbourne University, Australia, in 2012. He is currently an assistant professor and director of the High Performance Cloud Computing (HPCC) Laboratory, School of Computing and Informatics at University of Louisiana Lafayette, USA. His research focus is on Distributed and Cloud computing including heterogeneity, virtualization, resource allocation, and security.



**Magdy Bayoumi** received the BSc and MSc degrees in electrical engineering from Cairo University, Egypt, the MSc degree in computer engineering from Washington University, St. Louis, and the PhD degree in electrical engineering from the University of Windsor, Ontario. He was the vice president for Conferences of the IEEE Circuits and Systems (CAS) Society. He is the recipient of the 2009 IEEE Circuits and Systems Meritorious Service Award and the IEEE Circuits and Systems Society 2003 Education Award.

**Nain-Feng Tzeng** (M86-SM92-F10) received the PhD degree in computer science from the University of Illinois at Urbana-Champaign. Since 1987, he has been with Center for Advanced Computer Studies, University of Louisiana at Lafayette, where he is currently a professor. He was on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, 1998-2001, and the *IEEE Transactions on Computers*, 1994-1998. He is a fellow of the IEEE.

**Rajkumar Buyya** is a professor of Computer Science and Software Engineering, Future fellow of the Australian Research Council, and director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, at the University of Melbourne, Australia. He is one of the highly cited authors in computer science and software engineering worldwide. Microsoft Academic Search Index ranked as #1 author in the world (2005-2016) for both field rating and citations evaluations in the area of distributed and parallel computing. He is a fellow of IEEE