Developing Cloud Software

Algorithms, Applications, and Tools

Edited by

Ivan Porres
Tommi Mikkonen
Adnan Ashraf

TUCS General Publication
No 60, October 2013
ISBN 978-952-12-2952-7
ISSN 1239-1905

4 Proactive Virtual Machine Allocation for Video Transcoding in the Cloud

Fareed Jokhio, Adnan Ashraf, Tewodros Deneke, Sébastien Lafond,
Ivan Porres, and Johan Lilius
Department of Information Technologies
Åbo Akademi University, Turku, Finland
Email: {fjokhio, aashraf, tdeneke, slafond, iporres, jolilius}@abo.fi

Abstract–Video transcoding refers to the process of converting a digital video from one compressed format to another. It is a compute-intensive operation. Therefore, transcoding of a large number of simultaneous video streams requires a large-scale distributed system. Moreover, to handle different load conditions in a cost-efficient manner, the distributed system should be dynamically scalable. Infrastructure as a Service (IaaS) clouds currently offer computing resources, such as Virtual Machines (VMs), under the pay-per-use business model, which can be used to create a dynamically scalable cluster of video transcoding servers. Determining the number of VMs to provision for a dynamic cluster is still an open problem. In this chapter, we present a proactive VM allocation approach to scale video transcoding service on a given IaaS cloud. For better resource utilization, quality of service, and cost-efficiency, we use video segmentation at the Group of Pictures (GOP) level. The proposed approach is demonstrated in discrete-event simulations and an experimental evaluation involving different load patterns. We also present a prototype implementation of a distributed video transcoder based on the message passing programming model and a dynamic load balancing approach.

Keywords-Cloud computing, resource allocation, video transcoding.

4.1 Introduction

The use of multimedia content is common in our life. It may consist of digital images, videos, voice, or animation. The use of video is no longer limited to TV-channels or cinema theaters. There are several user-friendly and inexpensive devices available such as cell phones, digital cameras, which can be used to capture, manipulate and store digital videos. Electronics devices such as digital computers are used to process video data very efficiently. Video production is common nowadays and a large number of digital videos are uploaded on YouTube[1] and other video hosting sites. To store and transmit a digital video in a cost-efficient manner, video compression is used. Video compression is a mature field and several video coding standards are available such as MPEG-4 [36] and H.264 [37]. However, end-user devices do not support all video compression formats. Therefore, an unsupported video format needs to be converted into another format, which is supported by the target device. Converting a compressed video into another compressed video is known as video transcoding [35]. There are different types of video transcoding, such as bit-rate reduction in order to meet network bandwidth availability, resolution reduction for display size adoption, temporal transcoding for frame rate reduction, and error resilience transcoding for insuring high Quality of Service (QoS) [13], [39].

Video transcoding involves decoding and encoding processes. It is a compute-intensive process, usually performed at the server-side. It may be done in real-time or in batch processing. However, for an on-demand video streaming service, if the required video is not available in the desired format, the transcoding needs to be done on-the-fly in real-time. One of the main challenges of a real-time video transcoding operation is that it must avoid over and underflow of the output video buffer, which temporarily stores the transcoded videos at the server-side. The overflow occurs if the video transcoding rate exceeds the video play rate and the capacity of the buffer. Likewise, the buffer underflow may occur when the play rate exceeds the transcoding rate, while the buffer does not contain enough frames either, to avoid the underflow situation.

Video transcoding of a large number of video streams requires a large-scale cluster-based distributed system. Moreover, to handle varying amounts of load in a cost-efficient manner, the cluster should be dynamically scalable. Cloud computing provides theoretically infinite computing resources, which can be provisioned in an on-demand fashion under the pay-per-use business model [5]. Infrastructure as a Service (IaaS) clouds currently offer

[1]http://www.youtube.com/

computing resources, such as Virtual Machines (VMs), storage, and network bandwidth [32], which can be used to create a dynamically scalable cluster of video transcoding servers.

In a cloud environment, a video transcoding operation can be performed in several different ways. For example, it is possible to map an entire video stream on a dedicated VM. However, it requires a large number of VMs to transcode several simultaneous streams. Moreover, transcoding of High Definition (HD) video streams can take more time, which may violate the client-side QoS requirements of desired play rate [10]. Another approach is to split the video streams into smaller segments and then transcode them independently of one another [21]. In this approach, one VM can be used to transcode a large number of video segments belonging to different video streams. Moreover, video segments of one particular stream can be transcoded on multiple VMs.

In this chapter, we present prediction-based dynamic resource allocation and deallocation algorithms to scale video transcoding service on a given IaaS cloud in a horizontal fashion. The proposed algorithms allocate and deallocate VMs to a dynamically scalable cluster of video transcoding servers. We use a two-step load prediction method [2], which predicts a few steps ahead in the future to allow proactive resource allocation. For cost-efficiency, we share VM resources among multiple video streams. The sharing of the VM resources is based on the video segmentation, which splits the streams into smaller segments that can be transcoded independently of one another. The proposed approach is evaluated in two simulation-based experiments involving two different load patterns. The results show that it provides cost-efficient resource allocation for a large number of simultaneous streams while avoiding over and underflow of the output video buffer.

We also present the implementation of a distributed transcoder based on the message passing programming model on top of an Amazon Elastic Compute Cloud (EC2)[2] cluster. Among different methods of distributed computing we have chosen to use Message Passing Interface (MPI)[3] because of its maturity, support, open source nature, scalability, and ease of use. MPI is a message passing interface for Multiple Instruction Multiple Data (MIMD)[4] distributed memory concurrent computers and workstations [27]. In this programming model, a set of tasks that use their own local memory during computation can be performed on the same physical machine as well as across an arbitrary number of machines. Tasks exchange data through

[2]http://aws.amazon.com/ec2/

[3]http://www.mcs.anl.gov/research/projects/mpi/

[4]http://www.springerreference.com/docs/html/chapterdbid/311449.html

communication channels by sending and receiving messages. This means that data transfer usually requires cooperative operations to be performed by each process. MPI provides a library consisting of a set of basic functions that can be used to write parallel and distributed programs. The programmer is thus responsible and free to express all parallelism involved [16], [27]. To deploy the distributed video transcoding framework in a cloud, we selected StarCluster[5], which is used for cluster management. It is an open source cluster computing tool-kit for Amazon EC2. The main purpose of StarCluster is to automate overall process of building a cluster of VMs in Amazon EC2, which can be used for parallel and distributed computing.

We proceed as follows. In Section 4.2, we describe video bit stream structure. Section 4.3 presents the system architecture of the proposed VM allocation approach. Video segmentation is described in Section 4.4. Section 4.5 presents the proposed proactive VM allocation algorithms. Our load prediction approach is detailed in Section 4.6. In Section 4.7, we introduce our prototype implementation of MPI-based distributed video transcoder and present our dynamic load balancing algorithm for video transcoding in cloud computing. Section 4.8 presents simulation results of the proposed VM allocation approach, while Section 4.9 presents results of the prototype implementation of our MPI-based distributed video transcoder and our dynamic load balancing algorithm. Section 4.10 discusses important related works and Section 4.11 presents conclusion.

4.2 Video Bit Stream Structure

A Video stream is made up of different types of compressed frames. The video frames are organized into logical groups known as Group of Picturess (GOPs) and sequences. As shown in Figure 4.1, a video sequence comprises a sequence header and one or more GOPs. A GOP consists of different types of frames such as I (intra), P (predicted), and B (bi-directional predicted) containing all necessary information required to decode them.

Both I and P frames can be used as reference frames. However, an I frame is an independent reference frame that does not require any other reference frame in the decoding process. I-Frames have the highest quality, but have the largest size. A GOP starts with an I frame, which is followed by a number of P and B frames. Both P and B frames always require reference frames in the decoding process [36]. The P-frames use information from the previous I-Frames or P-frames to compress the frame. The B-frames

[5]http://star.mit.edu/cluster/

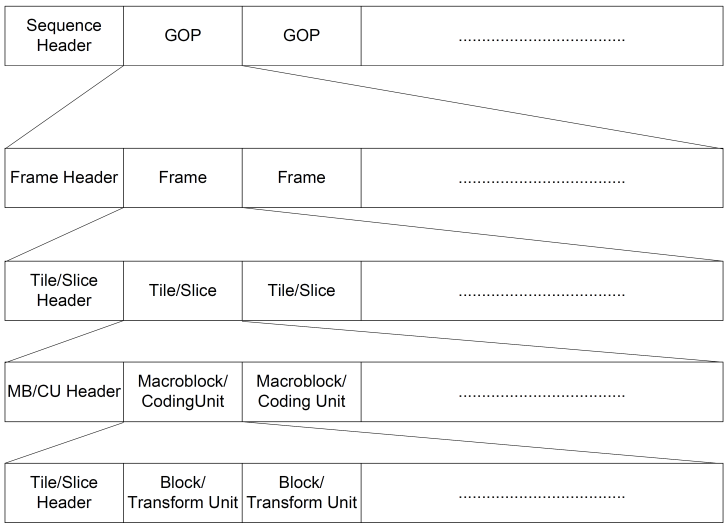

Figure 4.1: Structure of a video down to block level [36]

use information from both the previous and next *I*-Frame or *P*-Frames. *B*-Frames are the smallest in size, but have the least quality.

Frame prediction is used to reduce the size of frames by taking into account the previous and future frames to further improve the efficiency of compressed frames. Furthermore, a frame itself may be divided into smaller blocks or slices with each slice being able to use frame prediction independently of other slices.

A GOP in MPEG-4 can have from 0 to 3 *B* frames between successive *P* frames. Usually, it is 2. The distance between successive *I* frames is *N*, which includes both *P* and *B* frames. In many cases, the value of *N* is 12, but it can be any value between 1 and a few hundreds.

Different kinds of frames also require a different amount of memory. Typically, *I* frames require the largest number of bytes to represent images, for example 300 Kilobytes (KB). The *P* frames require less memory, for example 160 KB. The *B* frames require even less, for example 40 KB. The resolution of a video stream is measured in pixels and is usually written as horizontal x vertical, such as (1280 x 720). The frame-rate of a video is the number of frames displayed in a second, usually 24 to 30 frames per seconds (fps).

Video coding standards use a YCbCr color space [12] with three different components called *luma* (Y) and *chroma* (Cb, Cr). The *luma* component represents brightness while *chroma* components represents the color information.

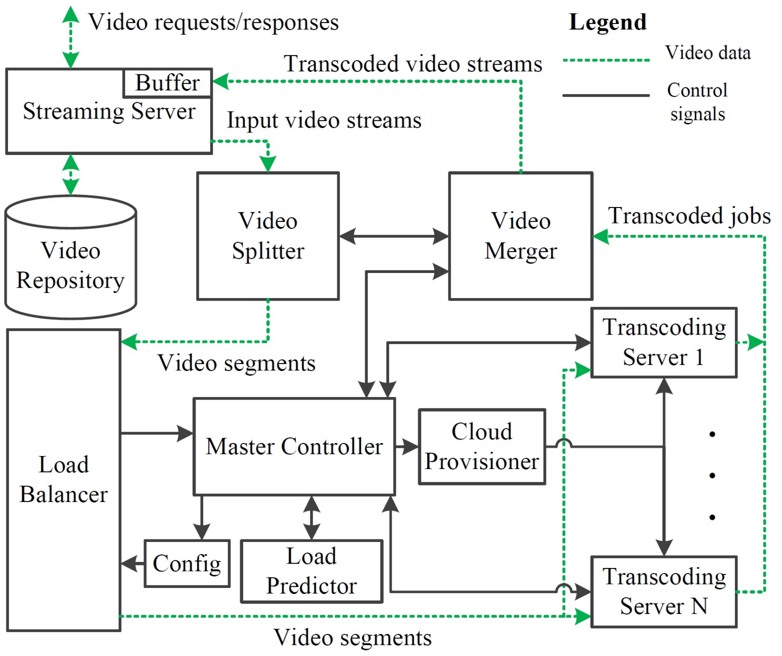

Figure 4.2: System architecture of the proactive VM allocation approach

A picture is usually divided into smaller parts termed as *macroblocks* or *coding units*. The *macroblocks* or *coding units* are the basic building blocks of video standards and the decoding of frames is performed at this level. To compress a frame, different techniques are used at *macroblock* or *coding unit* level such as, frame prediction in either spatial or temporal direction. The smaller partitions of the *macroblocks* are termed as *blocks* or *transform units*. The *transform coding* is performed at block level in various video coding standards.

4.3 System Architecture

The system architecture of the distributed video transcoding in cloud computing environment is shown in Figure 4.2. It consists of a *streaming server*, a *video splitter*, a *video merger*, a *video repository*, a dynamically scalable cluster of *transcoding servers*, a *load balancer*, a *master controller*, a *load predictor*, and a *cloud provisioner*.

In our system, the end-users or clients may send requests for videos. These requests arrive at the *streaming server*. The *streaming server* provides the video streaming service. Streaming is a general term which means that the data being transfered from one location to another can be used immediately. In case of video streaming, a video is decoded and played as soon as enough data has been transferred[6].

To transfer multimedia contents (video and audio) over the Internet by using streaming services such as Video Desk[7], media streaming technology is used, which can deliver the media contents in real-time. At the user end, parts of a video are downloaded, decoded, and played. The video playing and downloading happen at the same time. A buffer is used to place additional video contents from the streaming server. The overall process is invisible to the viewer. The streaming server works as a media servers which sends streamed video to users connected through different networks. Since the main focus of this research work is on video transcoding, we assume that the *streaming server* is not a bottleneck.

The video streams in certain compressed formats are stored in the *video repository*. The compressed videos can be either source videos or transcoded videos. The source videos are the original videos and transcoded videos are obtained from source videos after transcoding. The transcoded videos are stored as long as it is cost-efficient to store them. The *streaming server* accepts video requests from users and checks if the required video is available in the *video repository*. If it finds the video in the desired format and resolution, it starts streaming the video. However, if it finds that the requested video is stored only in another format or resolution than the one desired by the user, it sends the video for segmentation and subsequent transcoding. Then, as soon as it receives the transcoded video from the *video merger*, it starts streaming the video.

The *video splitter* splits the video streams into smaller segments called jobs, which are placed into the job queue. Due to inter-dependency among different types of frames, video segmentation (splitting) can be performed at certain points only. When splitting the video, the main issue is to perform the segmentation of source video so that parts of video can be distributed among transcoding servers. Section 4.4 discusses video segmentation in more detail.

The *load balancer* employs a task assignment policy, which distributes load on the transcoding servers. In other words, it decides when and to which *transcoding server* a transcoding job should be routed. It maintains a configuration file, which contains information about *transcoding servers*

[6]http://www.wimpyplayer.com/docs/faqs/docs/general_streaming_definition.html
[7]http://www.videodesk.net/

that perform the transcoding operations. As a result of dynamic resource allocation and deallocation operations, the configuration file is often updated with new information. The *load balancer* serves the jobs in First In, First Out (FIFO) order and has only one input queue. The *load balancer* implements the *shortest queue waiting time* policy, which selects a *transcoding server* with the shortest queue waiting time. Our dynamic load balancing algorithm is presented in Section 4.7.

The actual transcoding is performed by the transcoding servers. They get compressed video segments, perform the required transcoding operations, and return the transcoded video segments for merging. A *transcoding server* runs on a dynamically provisioned VM. Each *transcoding server* processes one or more simultaneous jobs. When a transcoding job arrives at a *transcoding server*, it is placed into the server's queue from where it is subsequently processed.

The *master controller* implements prediction-based dynamic resource allocation and deallocation algorithms, as described in section 4.5. For load prediction, the *master controller* uses *load predictor*, which is elaborated in section 4.6. The *cloud provisioner* refers to the cloud provisioner in an IaaS cloud, such as the provisioner in Amazon EC2. It performs the actual lower level tasks of starting and terminating VMs. The *video merger* merges the transcoded jobs into video streams, which form video responses.

The system architecture is similar to our previous work on prediction-based dynamic resource allocation for video transcoding in cloud computing [22]. However, in [22] the load balancer implements the *shortest queue length* policy, while in this chapter it implements the *shortest queue waiting time* policy, which provides improved performance. The *shortest queue length* policy is based on the queue size alone. Therefore, it does not account for the execution time of individual jobs in the queue. Whereas, the *shortest queue waiting time* policy uses estimated execution time of individual jobs in the queue to calculate the waiting time of new arriving jobs at the server queue. In addition to the *shortest queue waiting time* policy, we introduce some important enhancements to our VM allocation algorithms.

4.4 Video Segmentation

Distributed computing allows to speedup the transcoding process while maintaining the same quality of video. Due to inter-dependency among different types of frames, video segmentation can be performed at certain points only. The main problem is to perform the segmentation of source video in such a way that parts of the video can be distributed among transcoding servers.

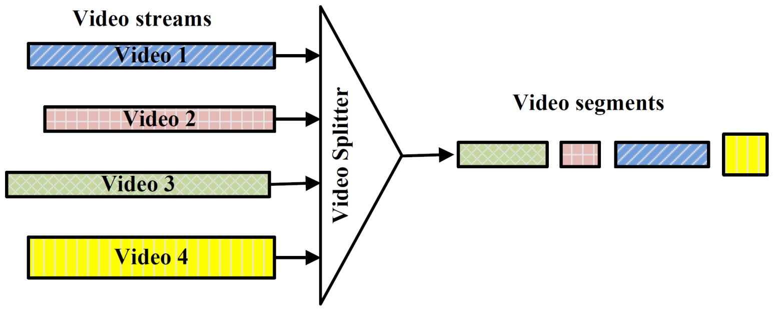

Figure 4.3: Video segmentation

Compressed video files contain different types of frames (I, P, B) which have different compression rates and inter-dependencies among them. Therefore, one can not split a given video at any particular frame or point. Among the frame types, an I-frame is an independent frame that can be decoded without any other reference frame. It is also used as a reference frame for other frames. In a given video, a sequence of frames that constitute an I-frame and a number of other B and P frames is called a GOP. GOPs are atomic units that can be transcoded independently of one another. Therefore, for efficient use of computing resources, we use video segmentation at the GOP level.

Video segmentation of four video streams is shown in Figure 4.3. The output of the video splitter consists of a number of jobs, where each job has at least one GOP. The video splitter tries to manage segmentation in such a way that each user gets a smooth video stream from the *streaming server*. It takes into account the transcoding time and the play time of the video segment. Once a video segment is sent for transcoding, the next segment of the same stream is sent after some delay.

The delay between two jobs during segmentation is based on the play time of a video segment and the number of transcoded video frames of the stream in the output buffer. If the transcoded frames are below certain predefined lower threshhold, the stream segmentation is performed with zero delay. However, if the transcoded frames are above the threshold, the delay for next video segment is set equal to the play time of previous job.

4.5 Proactive VM Allocation Algorithms

Under-utilization of computing servers in cloud computing is a common problem. Due to change in the load patterns at different times, there might be some over-provisioning of servers, which increases the overall cost [6], [31]. In addition, the over-utilization of resources is also not desirable due to high response times. Therefore, automatic VM allocation is essential for cost and performance efficiency [40]. In this section, the dynamic VM allocation and deallocation algorithms for video transcoding in the cloud are presented. For the sake of clarity, the concepts used in the algorithms and their notation are summarized in Table 4.1.

The algorithms implement proactive control, which uses a two-step prediction in which the load pattern of the system is tracked and then future load pattern is predicted. The *transcoding servers* are added and removed based on the predicted load and current throughput. Moreover, due to the VM provisioning delay, a fixed minimum number of transcoding servers is always maintained to provide an effective service. This is termed as the base capacity N_B.

On discrete-time intervals, the *master controller* obtains the play rate of all video streams, and sums up the play rates of streams, to get the total target play rate $PR(t_i)$. It then obtains the video transcoding rate from each *transcoding server* and calculates the total transcoding rate $TR(t_i)$. Moreover, for proactive VM allocation, it uses *load predictor* to predict the total transcoding rate of all *transcoding servers* $\hat{TR}(t_i)$ a few steps ahead in the future.

The algorithms are designed to be cost-efficient while minimizing potential oscillations in the number of VMs [38]. This is desirable because, in practice, provisioning of a VM takes a few minutes [8]. Therefore, oscillations in the number of VMs may lead to deteriorated performance. Moreover, since some contemporary IaaS providers, such as Amazon EC2, charge on hourly basis, oscillations will result in a higher provisioning cost. Therefore, the algorithms counteract oscillations by delaying new VM allocation operations until previous VM allocation operations have been realized [20]. Furthermore, for cost-efficiency, the deallocation algorithm terminates only those VMs whose renting period approaches its completion.

4.5.1 VM Allocation Algorithm

The VM allocation algorithm is given as Algorithm 4.1. The first two steps deal with the calculation of the target play rate $PR(t_i)$ of all streams and the total transcoding rate $TR(t_i)$ of all *transcoding servers*. The algorithm then obtains the predicted total transcoding rate $\hat{TR}(t_i)$ from the *load predictor*. Moreover, to avoid underflow of the output video buffer that temporarily stores transcoded jobs at the server-side, it considers the size of the output video buffer $B_S(t_i)$. If the target play rate exceeds the predicted transcoding

Table 4.1: Summary of concepts and their notation

$count_{over}(t_i)$	over allocation count at discrete-time t_i
$S(t_i)$	set of *transcoding servers* at t_i
$S_p(t_i)$	set of newly provisioned servers at t_i
$S_c(t_i)$	servers close to completion of renting period at t_i
$S_t(t_i)$	servers selected for termination at t_i
$PR(t_i)$	sum of target play rates of all streams at time t_i
$TR(t_i)$	total transcoding rate of all servers at time t_i
$\hat{TR}(t_i)$	predicted total transcoding rate at time t_i
$RT(s, t_i)$	remaining time of server s at t_i
$V(t_i)$	set of video streams at t_i
$N_P(t_i)$	number of servers to provision at t_i
$N_{P_Q}(t_i)$	number of servers to provision at t_i based on queue length
$N_T(t_i)$	number of servers to terminate at t_i
$getPR()$	get $PR(t_i)$ from video merger
$getTR(s)$	get transcoding rate of server s
$get\hat{TR}()$	get $\hat{TR}(t_i)$ from load predictor
$calN_P()$	calculate the value of $N_P(t_i)$
$calQN_P()$	calculate the value of $N_{P_Q}(t_i)$ based on queue length
$calN_T()$	calculate the value of $N_T(t_i)$
$calRT(s, t_i)$	calculate the value of $RT(s, t_i)$
$delay()$	delay function
$provision(n)$	provision n servers
$select(n)$	select n servers for termination
$sort(S)$	sort servers S on remaining time
$terminate(S)$	terminate servers S
C_T	over allocation count threshold
RT_U	remaining time upper threshold
RT_L	remaining time lower threshold
$MAXQL_{UT}$	Maximum Queue length upper threshold
B_L	buffer size lower threshold in megabytes
$B_S(t_i)$	size of the output video buffer in megabytes
B_U	buffer size upper threshold in megabytes
N_B	number of servers to use as base capacity
$startUp$	server startup delay
$avgQJobs$	average number of jobs in a server Queue
$jobCompletion$	job completion delay

rate while the buffer size $B_S(t_i)$ falls below its lower threshold B_L, the algorithm chooses to allocate resources by provisioning one or more VMs. The number of VMs to provision $N_P(t_i)$ is calculated as follows

$$N_P(t_i) = \left\lceil \frac{PR(t_i) - \hat{TR}(t_i)}{\frac{TR(t_i)}{|S(t_i)|}} \right\rceil \tag{4.1}$$

The algorithm then provisions $N_P(t_i)$ VMs, which are added to the cluster of *transcoding servers*. To minimize potential oscillations due to unnecessary VM allocations, the algorithm adds a delay for the VM startup time. Furthermore, it ensures that the total number of VMs $|S(t_i)|$ does not exceed the total number of video streams $|V(t_i)|$. The algorithm adjusts the number of VMs to provision $N_P(t_i)$ if $|S(t_i)| + N_P(t_i)$ exceeds $|V(t_i)|$. This is desirable because the transcoding rate of a video on a single VM is usually higher than the required play rate.

The VM allocation algorithm also takes into account the current load on servers. It checks the queue lengths of servers and if the average number of jobs in the queues is above a predefined maximum upper threshold, it provisions one or more servers. The number of VMs to provision $N_{P_Q}(t_i)$ is calculated as follows

$$N_{P_Q}(t_i) = \left\lceil \frac{avgQJobs}{MAXQL_{UT}} \right\rceil \tag{4.2}$$

4.5.2 VM Deallocation Algorithm

The VM deallocation algorithm is presented in Algorithm 4.2. The main objective of the algorithm is to minimize the VM provisioning cost, which is a function of the number of VMs and time. Thus, it terminates any redundant VMs as soon as possible. Moreover, to avoid overflow of the output video buffer, it considers the size of the output video buffer $B_S(t_i)$. After obtaining the target play rate $PR(t_i)$ and the predicted total transcoding rate $\hat{TR}(t_i)$, the algorithm makes a comparison. If $\hat{TR}(t_i)$ exceeds $PR(t_i)$ while the buffer size $B_S(t_i)$ exceeds its upper threshold B_U, it may choose to deallocate resources by terminating one or more VMs. However, to minimize unnecessary oscillations, it deallocates resources only when the buffer overflow situation persists for a predetermined minimum amount of time.

In the next step, the algorithm calculates the remaining time of each *transcoding server* $RT(s, t_i)$ with respect to the completion of the renting period. It then checks if there are any *transcoding servers* whose remaining time is less than the predetermined upper threshold of remaining time RT_U and more than the lower threshold of remaining time RT_L. The objective is to terminate only those servers whose renting period is close to completion, while excluding any servers that are extremely close to the completion of their renting period and therefore it is not cost-efficient to terminate them before the start of the next renting period. If the algorithm finds at least one

Algorithm 4.1. VM allocation algorithm

```
1:  while true do
2:      $N_P(t_i) := 0, N_{P_Q}(t_i) := 0$
3:      $PR(t_i) := getPR()$
4:      $TR(t_i) := 0$
5:      for $s \epsilon S(t_i)$ do
6:          $TR(t_i) := TR(t_i) + getTR(s)$
7:      end for
8:      $\hat{TR}(t_i) := get\hat{TR}(TR(t_i))$
9:      if $\hat{TR}(t_i) < PR(t_i) \wedge B_S(t_i) < B_L$ then
10:         $N_P(t_i) := calN_P()$
11:     end if
12:     if $avgQJobs > MAXQL_{UT}$ then
13:         $N_{P_Q}(t_i) := calQN_P()$
14:     end if
15:     $N_P(t_i) := N_P(t_i) + N_{P_Q}(t_i)$
16:     if $|S(t_i)| + N_P(t_i) > |V(t_i)|$ then
17:         $N_P(t_i) := |V(t_i)| - |S(t_i)|$
18:     end if
19:     if $N_P(t_i) \geq 1$ then
20:         $S_p(t_i) := provision(N_P(t_i))$
21:         $S(t_i) := S(t_i) \cup S_p(t_i)$
22:         $delay(startUp)$
23:     end if
24: end while
```

such server $S_c(t_i)$, it calculates the number of servers to terminate $N_T(t_i)$ as

$$N_T(t_i) = \left\lceil \frac{\hat{TR}(t_i) - PR(t_i)}{\frac{TR(t_i)}{|S(t_i)|}} \right\rceil - N_B \tag{4.3}$$

Then, it sorts the *transcoding servers* in $S_c(t_i)$ on the basis of their remaining time, and selects the servers with the lowest remaining time for termination. The rationale of sorting of servers is to ensure cost-efficiency by selecting the servers closer to completion of their renting period. A VM that has been selected for termination might have some pending jobs in its queue. Therefore, it is necessary to ensure that the termination of a VM does not abandon any jobs in its queue. One way to do this is to migrate all pending jobs to other VMs and then terminate the VM [8]. However, since transcoding of video segments takes relatively less time to complete, it is more reasonable

Algorithm 4.2. VM deallocation algorithm

```
1: while true do
2:   $PR(t_i) := getPR()$
3:   $TR(t_i) := 0$
4:   for $s \epsilon S(t_i)$ do
5:     $TR(t_i) := TR(t_i) + getTR(s)$
6:   end for
7:   $\hat{TR}(t_i) := get\hat{TR}(TR(t_i))$
8:   if $\hat{TR}(t_i) > PR(t_i) \wedge B_S(t_i) > B_U \wedge count_{over}(t_i) > C_T$ then
9:     for $s \epsilon S(t_i)$ do
10:      $RT(s, t_i) := calRT(s, t_i)$
11:    end for
12:    $S_c(t_i) := \{\forall s \epsilon S(t_i) | RT(s, t_i) < RT_U \wedge RT(s, t_i) > RT_L\}$
13:    if $|S_c(t_i)| \geq 1$ then
14:      $N_T(t_i) := calN_T()$
15:      $N_T(t_i) := min(N_T(t_i), |S_c(t_i)|)$
16:      if $N_T(t_i) \geq 1$ then
17:        $sort(S_c(t_i))$
18:        $S_t(t_i) := select(N_T(t_i))$
19:        $S(t_i) := S(t_i) \setminus S_t(t_i)$
20:        $delay(jobCompletion)$
21:        $terminate(S_t(t_i))$
22:      end if
23:    end if
24:  end if
25: end while
```

to let the jobs complete their execution without requiring them to migrate and then terminate a VM when there are no more running and pending jobs on it. Therefore, the deallocation algorithm terminates a VM only when the VM renting period approaches its completion and all jobs on the server complete their execution. Finally, the selected servers are terminated and removed from the cluster.

4.6 Load Prediction

The existing load prediction models for web-based systems, such as [2], [3], [34], can be adapted to predict transcoding rate of the *transcoding servers* a few steps ahead in the future. Andreolini and Casolari [2] proposed a two-step approach to predict future load behavior under real-time constraints. The approach involves load trackers that provide a representative view of the load behavior to the load predictors, thus achieving two steps.

A load tracker (LT) filters out noise in the raw data to yield a more regular view of the load behavior [2]. It is a function $LT(\overrightarrow{S_n}(t_i)) : \mathbb{R}^n \rightarrow \mathbb{R}$, which inputs a measure s_i monitored at time t_i, and a set of previously collected n measures, that is $\overrightarrow{S_n}(t_i) = (s_{i-n}, ..., s_i)$, and provides a representation of the load behavior l_i at time t_i [2]. A sequence of LT values yields a regular view of the load behavior. There are different classes of LTs, such as simple moving average (SMA), exponential moving average (EMA), and cubic spline (CS) [3]. More sophisticated (time-series) models often require training periods to compute the parameters and/or off-line analyses [2]. Likewise, the linear (auto) regressive models, such as ARMA and ARIMA, may require frequent updates to their parameters [2], [34]. Therefore, in our approach [7], [22], the *load predictor* implements an LT based on the EMA model, which limits the computation delay without incurring oscillations and computes an LT value for each measure with high prediction accuracy.

The load predictor (LP) is a function $LP_h(\overrightarrow{L_q}(t_i)) : \mathbb{R}^q \rightarrow \mathbb{R}$, which inputs a sequence of LT values $\overrightarrow{L_q}(t_i) = l_{i-q}, ..., l_i$ and outputs a predicted future value at time t_{i+h}, where $h > 0$ [2]. The LP is characterized by the prediction window h and the past time window q. Andreolini and Casolari [2] and Saripalli et al. [34] used linear regression of only two LT values, which are the first l_{i-q} and the last l_i values in the past time window. Ashraf et al. [7] and Jokhio et al. [22] used simple linear regression model [28], which takes into account all LT values $\overrightarrow{L_q}(t_i)$ in the past time window. The LP of the LT in this approach is based on a straight line defined as

$$l = \gamma_0 + \gamma_1 t \tag{4.4}$$

where γ_0 and γ_1 are called regression coefficients, which can be estimated at runtime based on the LT values [7], [28].

4.7 MPI-Based Distributed Video Transcoder

Our distributed video transcoder is based on the open source FFMPEG[8] library. We have modified the original transcoder to execute on multiple machines. The extension is based on MPI, where a set of processes each running a single transcoder, are made to collaborate and transcode a set of streams more efficiently. The MPI programming model allows programmers to explicitly specify the parallel processing in a given system. Unlike other

[8]http://www.ffmpeg.org/

frameworks like Hadoop, which is designed for a specific set of problems(i.e. batch processing), MPI provides programmers with flexibility in expense of some programming overhead.

Figure 4.4 shows the architecture of the MPI-based video transcoder for multiple streams. Each active incoming stream is segmented continuously and stored in a job queue of one of the transcoding servers. Load balancing is performed dynamically depending on an estimated queue waiting time of a segment on each transcoding server. The estimated waiting time of a new segment on a given transcoding server is calculated by dividing the number of frames that are currently in its queue with the average transcoding rate of the server (see Algorithm 4.3). The *manager* sends the next segment to a transcoding server with the smallest estimated queue waiting time. A header containing the stream ID, segment ID, a transcoding parameter, and number of frames in the segment is attached to each segment. This header is used to identify each segment in the system and make load balancing decisions. The total number of transcoding servers is decided by the *manager*. In our distributed transcoder, every server has its own ID and the work is routed according to these IDs. In Figure 4.4, the ID of the *manager* is 0. It implements the proposed dynamic load balancing algorithm. Moreover, it contains the *video splitter* and the *video merger*. The manager invokes *video splitter* before sending transcoding jobs to the transcoding servers. Each transcoding server takes a segment from its queue, parses the segment header to get the transcoding parameters, transcodes the segment, and sends back the transcoded segment to the *manager*. The *manager* then invokes the *video merger* to merge the transcoded segments into output streams. The impact of the dynamic load balancing algorithm is compared with a static round-robin approach in Section 4.9.

4.8 Simulation Results of VM Allocation

Software simulations are often used to test and evaluate new algorithms involving complex environments [11]. We have developed a discrete-event simulation for the proposed VM allocation approach. The simulation is written in the Python programming language and is based on the SimPy simulation framework [26].

4.8.1 Experimental Design and Setup

We considered two different synthetic load patterns in two separate experiments. Load pattern 1 in experiment 1 consists of two load peaks, while

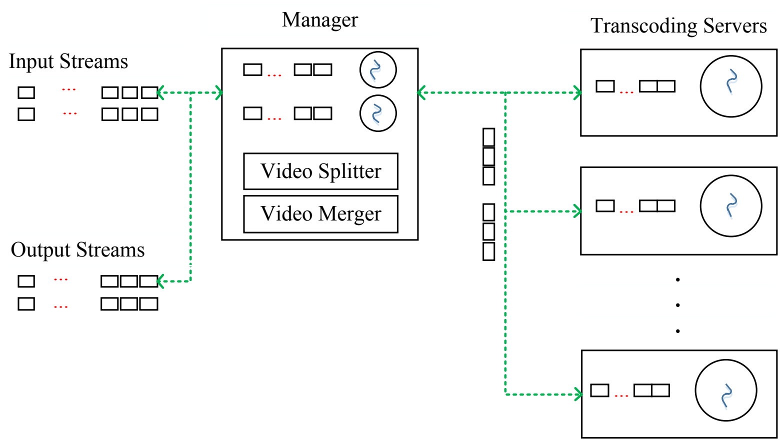

Figure 4.4: MPI-based distributed video transcoder

Algorithm 4.3. Dynamic load balancing

```
1: while true do
2:   $selected\_server = transcoding\_servers.get(0)$
3:   $smallest\_queue\_time = TIME\_MAX$
4:   for $transcoding\_server$ in $transcoding\_servers$ do
5:     $queue\_time = \frac{transcoding\_server->frames}{transcoding\_server->fps}$
6:     if $queue\_time < smallest\_queue\_time$ then
7:       $smallest\_queue\_time = queue\_time$
8:       $selected\_server = transcoding\_server$
9:     end if
10:  end for
11:  $segment = queue->take()$
12:  $send(segment, selected\_server)$
13: end while
```

load pattern 2 in experiment 2 has six load peaks. For simplicity, the renting period was assumed to be 600 seconds. The remaining time upper threshold RT_U was 60 seconds, while the remaining time lower threshold RT_L was 12 seconds. The Load Tracker (LT) and lp parameters were as follows: $n = 15$, $q = 30$, and $h = 120$.

The experiments used both Standard Definition (SD) and HD video streams. At the time of writing this book chapter, 10% of YouTube's videos are available in HD, while YouTube has more HD content than any other video hosting site [1]. However, the ratio of HD versus SD is expected to increase in the near future. Therefore, the load generation assumed 70% SD and 30% HD video streams. The video segmentation was performed at the GOP level. The segmentation produced video segments, which were sent to the *transcoding servers* for execution. For HD videos, the average size of a video segment was 75 frames with a standard deviation of 7 frames. Likewise, for SD videos, the average size of a segment was 250 frames with a standard deviation of 20 frames. The total number of frames in a video stream was in the range of 15000 to 18000.

The desired play rate for a video stream is often fixed: 30 fps for SD videos and 24 fps for HD videos. Whereas, the transcoding rate depends on the video contents, such as, frame resolution, type of video format, type of frames, and contents of blocks. Different transcoding mechanisms also require different times.

In our experiments, the maximum transcoding rate for SD videos was assumed to be four times of its play rate. We further assumed that the transcoding rate is always higher than the play rate of all video streams. Similarly, the minimum transcoding rate for SD videos was assumed to be double of its play rate. Since HD videos require more computation, the maximum transcoding rate for an HD video was assumed to be double of the play rate, with the minimum transcoding rate at 1.5 times the play rate.

The objective of experiment 1 was to simulate a relatively normal load. It was designed to generate a load representing a maximum of 200 simultaneous video streams in two different load peaks. In the first peak, the streams were ramped-up from 0 to 200, while adding a new stream every 20 seconds. After the ramp-up phase, the number of streams was maintained constant for 1 hour and then ramped-down to 100 streams.

The second peak ramped-up from 100 streams to 200 streams, while adding a new stream every 30 seconds. The ramp-up phase was followed by a similar constant phase as in the first peak. Then, the ramp-down phase removed all streams from the system.

Experiment 2 was designed to simulate the load pattern of a highly variable video demand. It generated a load representing a maximum of 280 simultaneous video streams consisting of six different load peaks. In the first peak, the streams were ramped-up from 0 to 170, while adding a new stream every 30 seconds. Then, in the second peak from 110 to 250, while adding a new stream every 20 seconds. Likewise, 210 to 280, 215 to 250, 120 to 200, and 100 to 170, respectively, in the third, fourth, fifth, and sixth peak. The stream ramp-up rate was 1 new stream per 30 seconds. Each ramp-up phase was followed by a ramp-down phase. Finally, the last ramp-down phase removed all streams from the system.

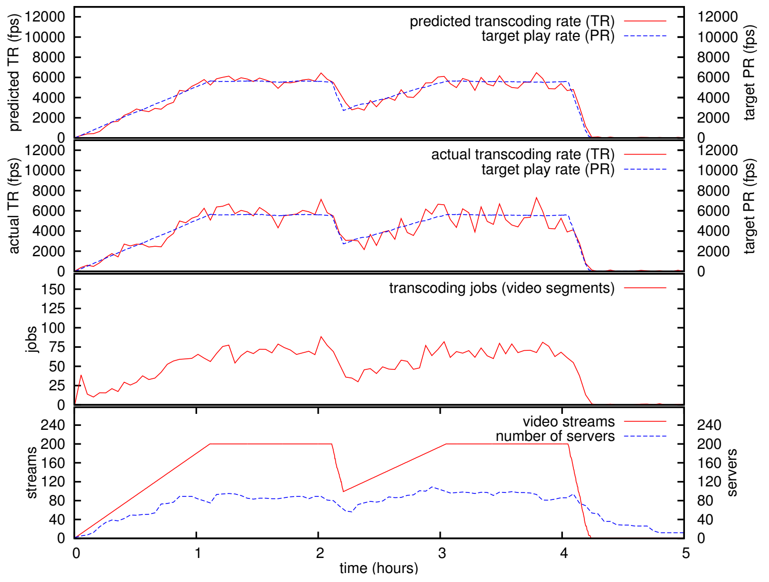

Figure 4.5: Experiment 1 results: relatively normal load

4.8.2 Results and Analysis

In both Figures 4.5 and 4.6, the number of servers plot shows dynamic VM allocation for the cluster of *transcoding servers*. The transcoding jobs plot represents the total number of jobs in the system at a particular time instance. It includes the jobs in execution at the *transcoding servers* and the jobs that were waiting in the queues. The target play rate plot shows the sum of target play rates of all video streams in the system. Likewise, the actual transcoding rate plot represents the total transcoding rate of all servers, while the predicted transcoding rate plot shows results of the load prediction. As described in Section 4.5, the VM allocation decisions were mainly based on the target play rate, the predicted transcoding rate, and the queue length of servers.

Figure 4.5 presents results from experiment 1. The results are also summarized in Table 4.2. Experiment 1 used a maximum of 93 *transcoding servers* for a maximum of 200 simultaneous streams. Moreover, a total of 4596 streams consisting of approximately 5×10^5 transcoding operations and 7×10^7 video frames were completed in 4 hours and 38 minutes. The results indicate that the resource allocation algorithms with the sharing of the VM resources among multiple video streams resulted in a reduced number

Table 4.2: Results from proactive VM allocation experiments. Experiment 1 uses relatively normal load, while experiment 2 uses highly variable load. The results include maximum number of servers used, maximum and average number of transcoding jobs or segments, maximum and average play rate (PR), and maximum and average transcoding rate (TR).

experiment	servers	$\text{jobs}_{\text{avg.}}$	jobs_{max}	$\text{PR}_{\text{avg.}}$	PR_{max}	$\text{TR}_{\text{avg.}}$	TR_{max}
1	109	44.84	95	3597.42 fps	5670 fps	3496.48 fps	7683.34 fps
2	150	40.74	122	3273.61 fps	7818 fps	3446.19 fps	9234.23 fps

of servers as compared with the number of video streams which reduces VM provisioning cost. The resource de-allocation algorithm takes into account the servers remaing renting time. A server is terminated only when it is near its completion of renting period, which avoids unnecessary oscillations in the number of VMs.

The results show that the actual transcoding rate was always close to the target play rate. This was desirable to avoid over and underflow of the output video buffer in the system, as discussed in Section 4.5. Although the actual transcoding rate was sometimes slightly above or below the target play rate, the proactive resource allocation helped to ensure that the cumulative number of transcoded frames was always greater than the cumulative number of played frames.

Figure 4.6 presents results from experiment 2. Table 4.2 also contains a summary of the results. It used a maximum of 120 *transcoding servers* for a maximum of 290 simultaneous streams. Moreover, a total of 7241 streams consisting of approximately 8×10^5 transcoding operations and 1×10^8 video frames were completed in 6 hours and 54 minutes. Although the number of streams was fluctuating rapidly, the algorithms provided a sustainable service with fewer VMs, while minimizing oscillations in the number of servers and avoiding the over and underflow of the output video buffer.

4.9 Evaluation of MPI-Based Distributed Video Transcoder

In this section, we describe the experimental setup, the results obtained, and their analysis from our prototype implementation of the MPI-based distributed video transcoder. The results are focused on the effect of load balancing algorithms on the utilization of transcoding servers. Therefore, provisioning of the transcoding servers is done statically before each experiment.

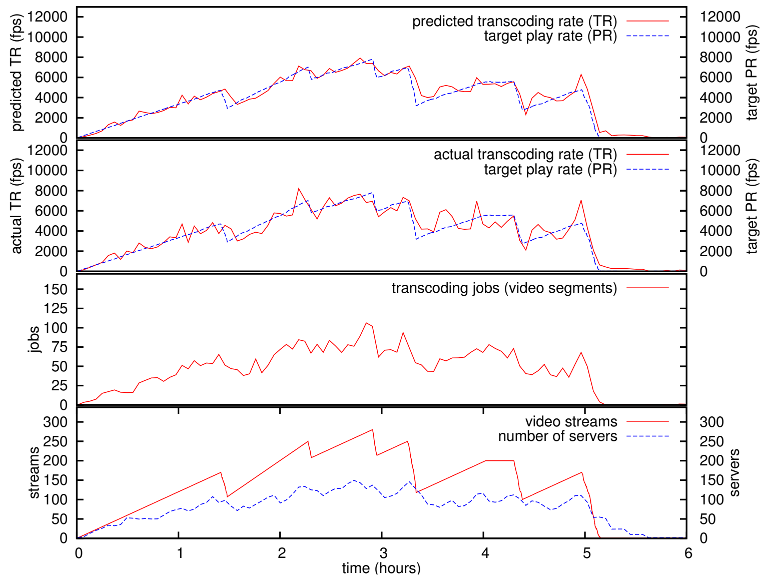

Figure 4.6: Experiment 2 results: highly variable load

4.9.1 Experimental Design and Setup

Our main task was to investigate performance in terms of total processing time of distributed video transcoding in cloud computing with dynamic load balancing. We setup a homogeneous and a heterogeneous test environment in the cloud using StarCluster. The StarCluster is an open source cluster computing tool-kit for Amazon EC2. It has been designed to automate and simplify the process of building, configuring, and managing clusters of VMs suited for distributed and parallel computing applications and systems on Amazon EC2.

The *homogeneous test cluster* consist of a *stream manager* and a maximum of 14 *transcoding servers*. The *stream manager* is assigned the task of splitting, merging, and scheduling of video segments. All the nodes in this cluster are *m1.small* instances from Amazon EC2. The *m1.small* instance is a VM with 1.7 GB memory and one virtual core running 32 bit Ubuntu 11.10 on AMD 2218HE. The *m1.small* instance is the default instance in the Amazon EC2 and is not optimized for anything in particular. Furthermore, the instances are selected from the same availability zone (i.e eu-west-1a). Running virtual cloud instances from the same availability zone ensures a more homogeneous network connection in the cluster.

The Heterogeneous test cluster consists of a maximum of 14 *transcoding servers* and a *stream manager*. Half nodes in the cluster are *m1.small* instances from Amazon EC2 cloud. The other half instances in the cluster are *c1.medium* instances. The *c1.medium* is a VM with $1.7GB$ memory and two virtual core running 32 bit Ubuntu 11.10 on Intel $E5-2650$ at 2 GHz. In contrast to the basic *m1.small* instance, the *c1.medium* instance has two cores and is optimized for speed. Furthermore, the instances are selected from two different availability zones (i.e eu-west-1a and eu-west-1c), which are located apart form each other. Running virtual cloud instances from different availability zone might lead to a more heterogeneous network connection in the cluster.

The aim of of doing the experiment on a heterogeneous clusters is motivated by the concept of job affinity [24], which states that some jobs may run significantly faster on nodes of a particular instance than others. Hence, it is important to know the job/instance type relationship and and design the load balancing algorithm accordingly.

To perform transcoding in the distributed environment, we have modified an existing open source transcoder FFMPEG with Message Passing Programming Model [27]. The first process with rank 0 is assigned the task of splitting, scheduling, and merging input video streams. The other processes with rank 1 to 14 are assigned the task of transcoding. The *manager* first splits incoming video streams at GOP level and uses the static or dynamic load balancing methods explained in Section 4.7. Depending on the scheduler decision, the *manager* sends video segments to a selected transcoding server's queue. The task of each transcoding server is to pick a task from their queue and perform the transcoding job till a termination signal is sent from the stream *manager*.

To perform different experiments in a cluster-based environment, we selected various video sequences having different number of frames. Characteristics of those video sequences such as length, size, total number of frames, and resolution are given in Table 4.3. All video sequences have a frame rate of 24 fps.

In this experiment we have restricted ourselves to using few streams as the focus is to only understand and compare the the effect of job load balancing approaches on the throughput of a distributed transcoding system.

4.9.2 Results and Analysis

We used both SD and HD video sequences in our experiments. Table 4.3 shows characteristics of video sequences, such as, size of video sequence in Mega Bytes, number of frames, and resolution. The total transcoding time

Table 4.3: Characteristics of video sequences

Video	Size	Frames	Resolution
Sintel HD	251 MB	21312	1280x720
Elephants Dream HD	162 MB	15690	1280x720
Big Buck Bunny HD	115 MB	14315	1280x720
Sintel SD	48 MB	21312	854x480
Elephants Dream SD	34 MB	15690	854x480
Big Buck Bunny SD	30 MB	14315	854x480

with different number of transcoding servers in a heterogeneous cloud cluster for Sintel HD, Elephants Dream HD, Big Buck Bunny HD, and for multiple streams with two HD and two SD simultaneous streams is shown in Figures 4.7a-d. Likewise, transcoding time with different number of servers in a homogeneous Cloud cluster for Sintel HD, Elephants Dreams HD, Big Buck Bunny HD, and for multiple streams with two HD and two SD simultaneous streams is shown in Figures 4.7e-h.

Figure 4.7a-d shows the total transcoding time for static round-robin and our proposed dynamic load balancing over a varying number of servers. In all cases, the performance gain accounts to about 45% except the case where there is only one transcoding node and the scheduling overhead matters significantly.

Figure 4.7e-h shows the result of applying the static and dynamic scheduling algorithms specified in Section 4.7 on a cluster of homogeneous transcoding servers. In this case, the performance gain from using the proposed dynamic load balancing algorithm only accounts to about $10-12\%$, except the cases when there is only one or two transcoding nodes resulting in a significant scheduling overhead. This result is also expected due to the fact that the experiment is done in a homogeneous platform and the performance gain is only due to the fact that video streams have different computational loads on different segments.

Figure 4.7d and Figure 4.7h show the results for the performance of the proposed dynamic scheduling algorithm against the static one when there are four simultaneous streams in the system. We selected the first two HD and two SD videos from Table 4.3 to test the multiple streams setup. As can be noticed from these figures, the gap between the performance of the dynamic and static schedulers slightly increased due to the non-homogeneity introduced by the Central Processing Unit (CPU) demand difference among HD and SD video streams.

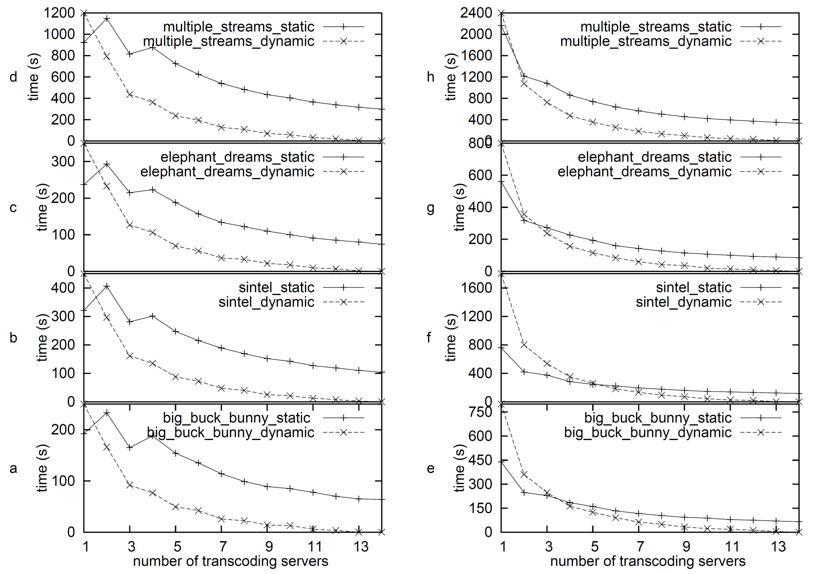

Figure 4.7: MPI-based distributed video transcoder results. Figures a-d show the total transcoding time with different number of transcoding servers in a heterogeneous cloud cluster for Sintel HD , Elephants Dream HD, Big Buck Bunny HD, and for multiple streams. Similarly, Figures e-h show the results in a homogeneous cloud cluster for Sintel HD, Elephants Dream HD, Big Buck Bunny HD, and for multiple streams.

4.10 Related Work

Distributed video transcoding with video segmentation was proposed in [21] and [23]. In these works, video segmentation was performed at the GOPs level. Jokhio et al. [21] presented bit rate reduction video transcoding using multiple processing units. The paper discussed computation, parallelization, and data distribution among computing units. In [23], different video segmentation methods were analyzed to perform spatial resolution reduction video transcoding. The paper compared three possible methods of video segmentation. In both papers, video transcoding was not performed in the cloud and the VM allocation problem was not addressed. In contrast, the main focus of this work is on VM allocation and deallocation algorithms. Huang et al. [19] presented a cloud-based video proxy to deliver transcoded videos for streaming. The main contribution of their work is a multilevel transcoding parallelization framework. They used Hallsh-based and Lateness-first mapping to optimize transcoding speed and to reduce transcoding jitters. The performance evaluation was done on a campus cloud testbed and the communication latency between cloud and video proxy was neglected. Li et al. [25] proposed *cloud transcoder*, which uses a compute cloud as an intermediate platform to provide transcoding service. Both papers do not discuss the VM allocation problem for video transcoding in cloud computing.

The existing works on dynamic VM allocation can be classified into two main categories: Plan-based approaches and control theoretic approaches. Plan-based approaches can be further classified into workload prediction approaches and performance dynamics model approaches. One example of the workload prediction approaches is Ardagna et al. [4], while TwoSpot [38], Hu et al. [18], Chieu et al. [14], Iqbal et al. [20] and Han et al. [17] use a performance dynamics model. Similarly, Dutreilh et al. [15], Pan et al. [29], Patikirikorala et al. [30], and Roy et al. [33] are control theoretic approaches. One common difference between all of these works and our proposed approach is that they are not designed specifically for video transcoding in cloud computing. In contrast, our proposed approach is based on the important VM allocation metrics for video transcoding service. Moreover, the proposed approach is cost-efficient as it uses fewer VMs for a large number of video streams and it counteracts possible oscillations in the number of VMs that may result in higher provisioning costs.

Ardagna et al. [4] proposed a distributed algorithm for managing Software as a Service (SaaS) cloud systems that addresses capacity allocation for multiple heterogeneous applications. The resource allocation algorithm takes into consideration a predicted future load for each application class and a predicted future performance of each VM, while determining possible Service-Level Agreement (SLA) violations for each application type. The main challenge in the prediction-based approaches is in making good prediction models that should provide high prediction accuracy under real-time constraints. For this, we use a two-step prediction approach, which limits the computation delay without incurring oscillations, while providing high prediction accuracy.

TwoSpot [38] aims to combine existing open source technologies to support web applications written in different programming languages. It supports hosting of multiple web applications, which are automatically scaled up and down in a horizontal fashion. However, the scaling down is decentralized, which may lead to severe random drops in performance. For example, when all controllers independently choose to scale down at the same time. Hu et al. [18] proposed a heuristic algorithm that determines the server allocation strategy and job scheduling discipline which results in the minimum number of servers. They also presented an algorithm for determining the minimum number of required servers, based on the expected arrival rate, service rate, and SLA. Chieu et al. [14] presented an approach that scales servers for a particular web application based on the number of active user sessions. The main problem with this approach is in determining suitable threshold values on the number of user sessions. Iqbal et al. [20] proposed an approach for adaptive resource provisioning for read intensive multi-tier web applications. Based on response time and CPU utilization metrics, the approach determines the bottleneck tier and then scales it up by provisioning a new VM. Scaling down is supported by checking for any over-provisioned resources from time to time. Han et al. [17] proposed a reactive resource allocation approach to integrate VM-level scaling with a more fine-grained resource-level scaling. In contrast, the proposed approach provides proactive VM allocation, where the VM allocation decisions are based on the important video transcoding metrics, such as video play rate and server transcoding rate.

Dutreilh et al. [15] and Pan et al. [29] used control theoretic models for designing resource allocation solutions for cloud computing. Dutreilh et al. presented a comparison of static threshold-based and reinforcement learning techniques. Pan et al. used Proportional-Integral (PI) controllers to provide QoS guarantees. Patikirikorala et al. [30] proposed a multi-model framework for implementing self-managing control systems for QoS management. Roy et al. [33] presented a look-ahead resource allocation algorithm based on the model predictive control. A common characteristic of the control theretic approaches is that they depend upon performance and dynamics of the underlying system. In contrast, the proposed approach does not require any knowledge about the performance and dynamics of the *transcoding servers*.

4.11 Conclusion

In this chapter, we presented proactive VM allocation algorithms to scale video transcoding service in a cloud environment. The proposed algorithms provide a mechanism for creating a dynamically scalable cluster of video transcoding servers by provisioning VMs from an IaaS cloud. The prediction of the future user load is based on a two-step load prediction method, which allows proactive VM allocation with high prediction accuracy under real-time constraints. For cost-efficiency, we used segmentation of video streams, which splits a stream into smaller segments that can be transcoded independently of one another. This helped us to perform video transcoding of multiple streams on a single server. The proposed VM allocation approach is demonstrated in a discrete-event simulation. The evaluation and analysis considered two different synthetic load patterns in two separate experiments. Experiment 1 used a relatively normal load, while experiment 2 used a highly variable load. The results show that the proposed approach provides cost-efficient VM allocation for transcoding a large number of video streams, while minimizing oscillations in the number of servers and avoiding over and underflow of the output video buffer.

We also presented a prototype implementation of a MPI-based distributed video transcoder and a dynamic load balancing algorithm for video transcoding in cloud computing. Experimental results from the MPI implementation show that the distributed transcoding approach along with the dynamic load balancing scheme decreases the total transcoding time up to 45% for a heterogeneous set of servers and up to 12% for homogeneous environments.

Future work includes implementing an admission controller to prevent transcoding servers from becoming overloaded. We have been currently working on a stream-based admission control approach for video transcoding in cloud computing [9]. Furthermore, a computation and storage trade-off strategy for video transcoding in cloud computing and using realistic load patterns for experimental evaluation are also part of our ongoing work.

Acknowledgments

This work was supported by an Amazon Web Services research grant. Fareed Jokhio and Adnan Ashraf were partially supported by the Foundation of Nokia Corporation and by doctoral scholarships from the Higher Education Commission (HEC) of Pakistan.

References

[1] *35 Mind Numbing YouTube Facts, Figures and Statistics Infographic.* 2012/05/23. URL: `http://www.jeffbullas.com/2012/05/23/35-mind-numbing-youtube-facts-figures-and-statistics-infographic/`.

[2] M. Andreolini and S. Casolari. “Load prediction models in web-based systems”. In: *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools.* valuetools ’06. New York, NY, USA: ACM, 2006. ISBN: 1-59593-504-5. DOI: `10.1145/1190095.1190129`.

[3] M. Andreolini, S. Casolari, and M. Colajanni. "Models and Framework for Supporting Runtime Decisions in Web-Based Systems". In: *ACM Transactions on the Web* 2.3 (2008), pp. 1–43. ISSN: 1559-1131. DOI: 10.1145/1377488.1377491.

[4] D. Ardagna et al. "Service Provisioning on the Cloud: Distributed Algorithms for Joint Capacity Allocation and Admission Control". In: *Towards a Service-Based Internet*. Ed. by E. Di Nitto and R. Yahyapour. Vol. 6481. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 1–12.

[5] M. Armbrust et al. "A view of cloud computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672.

[6] M. Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, 2009. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[7] A. Ashraf, B. Byholm, and I. Porres. "A Session-Based Adaptive Admission Control Approach for Virtualized Application Servers". In: *The 5th IEEE/ACM International Conference on Utility and Cloud Computing*. Ed. by C. Varela and M. Parashar. IEEE Computer Society, 2012, pp. 65–72.

[8] A. Ashraf et al. "Feedback Control Algorithms to Deploy and Scale Multiple Web Applications per Virtual Machine". In: *38th Euromicro Conference on Software Engineering and Advanced Applications*. Ed. by V. Cortellessa, H. Muccini, and O. Demirors. IEEE Computer Society, 2012, pp. 431–438.

[9] A. Ashraf et al. "Stream-Based Admission Control and Scheduling for Video Transcoding in Cloud Computing". In: *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CC-Grid)*. 2013.

[10] N. Bjork and C. Christopoulos. "Transcoder architectures for video coding". In: *Consumer Electronics, IEEE Transactions on* 44.1 (1998), pp. 88 –98. ISSN: 0098-3063. DOI: 10.1109/30.663734.

[11] R. N. Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50. ISSN: 1097-024X.

[12] D. Chai and A. Bouzerdoum. "A Bayesian approach to skin color classification in YCbCr color space". In: *TENCON 2000. Proceedings.* Vol. 2. 2000, 421–424 vol.2. DOI: `10.1109/TENCON.2000.888774`.

[13] S. F. Chang and A. Vetro. "Video Adaptation: Concepts, Technologies, and Open Issues". In: *Proceedings of IEEE* 93.1 (Jan. 2005), pp. 148–158. URL: `http://dx.doi.org/10.1109/JPROC.2004.839600`.

[14] T. C. Chieu et al. "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment". In: *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on.* 2009, pp. 281–286. DOI: `10.1109/ICEBE.2009.45`.

[15] X. Dutreilh et al. "From Data Center Resource Allocation to Control Theory and Back". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on.* 2010, pp. 410–417. DOI: `10.1109/CLOUD.2010.55`.

[16] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI, Portable Parallel Programming with the Message Passing Interface.* MIT Press.

[17] R. Han et al. "Lightweight Resource Scaling for Cloud Applications". In: *Cluster Computing and the Grid, IEEE International Symposium on* (2012), pp. 644–651.

[18] Y. Hu et al. "Resource provisioning for cloud computing". In: *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research.* CASCON '09. New York, NY, USA: ACM, 2009, pp. 101–111.

[19] Z. Huang et al. "CloudStream: Delivering high-quality streaming videos through a cloud-based SVC proxy". In: *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China.* IEEE, 2011, pp. 201–205. DOI: `http://dx.doi.org/10.1109/INFCOM.2011.5935009`.

[20] W. Iqbal et al. "Adaptive resource provisioning for read intensive multi-tier applications in the cloud". In: *Future Generation Computer Systems* 27.6 (2011), pp. 871–879. ISSN: 0167-739X.

[21] F. Jokhio et al. "Bit Rate Reduction Video Transcoding with Distributed Computing". In: *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on.* 2012, pp. 206 –212. DOI: `10.1109/PDP.2012.59`.

[22] F. Jokhio et al. “Prediction-Based Dynamic Resource Allocation for Video Transcoding in Cloud Computing”. In: *Parallel, Distributed and Network-Based Processing (PDP), 21st Euromicro International Conference on.* 2013.

[23] F. A. Jokhio et al. “Analysis of video segmentation for spatial resolution reduction video transcoding”. In: *Intelligent Signal Processing and Communications Systems (ISPACS), 2011 International Symposiuml.* 2011, 6 pp.

[24] G. Lee, B.-G. Chun, and H. Katz. “Heterogeneity-aware resource allocation and scheduling in the cloud”. In: *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing.* HotCloud’11. Portland, OR: USENIX Association, 2011, pp. 4–4. URL: `http://dl.acm.org/citation.cfm?id=2170444.2170448`.

[25] Z. Li et al. “Cloud Transcoder: Bridging the Format and Resolution Gap between Internet Videos and Mobile Devices”. In: *The 22nd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video.* ACM, 2012. ISBN: 978-1-4503-0752-9/12/06.

[26] N. Matloff. *A Discrete-Event Simulation Course Based on the SimPy Language.* University of California at Davis, 2006. DOI: `http://heather.cs.ucdavis.edu/~matloff/simcourse.html`. URL: `http://heather.cs.ucdavis.edu/~matloff/simcourse.html`.

[27] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Knoxville, TN: University of Tennessee, June 1995.

[28] D. Montgomery, E. Peck, and G. Vining. *Introduction to Linear Regression Analysis.* Wiley Series in Probability and Statistics. John Wiley & Sons, 2012. ISBN: 9780470542811.

[29] W. Pan et al. “Feedback Control-Based QoS Guarantees in Web Application Servers”. In: *High Performance Computing and Communications, 2008. HPCC ’08. 10th IEEE International Conference on.* 2008, pp. 328–334. DOI: `10.1109/HPCC.2008.106`.

[30] T. Patikirikorala et al. “A multi-model framework to implement self-managing control systems for QoS management”. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* SEAMS ’11. New York, NY, USA: ACM, 2011, pp. 218–227. ISBN: 978-1-4503-0575-4.

[31] M. Pawlish, A. S. Varde, and S. A. Robila. “Cloud computing for environment-friendly data centers”. In: *Proceedings of the fourth international workshop on Cloud data management.* CloudDB ’12. Maui, Hawaii, USA: ACM, 2012, pp. 43–48. ISBN: 978-1-4503-1708-5. DOI: 10.1145/2390021.2390030. URL: http://doi.acm.org/10.1145/2390021.2390030.

[32] J. Rhoton and R. Haukioja. *Cloud Computing Architected: Solution Design Handbook.* Recursive Press, 2011. ISBN: 9780956355614.

[33] N. Roy, A. Dubey, and A. Gokhale. “Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting”. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on.* 2011, pp. 500 –507. DOI: 10.1109/CLOUD.2011.42.

[34] P. Saripalli et al. “Load Prediction and Hot Spot Detection Models for Autonomic Cloud Computing”. In: *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on.* 2011, pp. 397 –402. DOI: 10.1109/UCC.2011.66.

[35] A. Vetro, C. Christopoulos, and H. Sun. “Video transcoding architectures and techniques: an overview”. In: *Signal Processing Magazine, IEEE* 20.2 (2003), pp. 18 –29. ISSN: 1053-5888. DOI: 10.1109/MSP.2003.1184336.

[36] J. Watkinson. *The MPEG Handbook: MPEG-1, MPEG-2, MPEG-4.* Broadcasting and communications. Elsevier/Focal Press, 2004. ISBN: 9780240805788.

[37] T. Wiegand, G. J. Sullivan, and A. Luthra. “Draft ITU-T recommendation and final draft international standard of joint video specification”. In: *Technical Report.* 2003.

[38] A. Wolke and G. Meixner. “TwoSpot: A Cloud Platform for Scaling out Web Applications Dynamically”. In: *Towards a Service-Based Internet.* Ed. by E. di Nitto and R. Yahyapour. Vol. 6481. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 13–24. ISBN: 978-3-642-17693-7.

[39] J. Xin, C.-W. Lin, and M.-T. Sun. “Digital Video Transcoding”. In: *Proceedings of the IEEE* 93.1 (2005), pp. 84 –97. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.839620.

[40] Y. Yazir et al. “Dynamic Resource Allocation in Computing Clouds Using Distributed Multiple Criteria Decision Analysis”. In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on.* 2010, pp. 91–98. DOI: 10.1109/CLOUD.2010.66.