CloudVideo: Elastic transcoding in the cloud

Authors: Jurre van Beek*, Henrik Hannemose†.
Instructors: Alexandru Iosup‡, Dick Epema‡
Lab assistant: Bogdan Ghit‡
*jurre@ch.tudelft.nl
†henrik@hannemose.dk
‡{a.iosup,d.h.j.epema,b.i.ghit}@tudelft.nl

***Abstract*—We present CloudVideo, a solution for transcoding videos in the cloud. While existing solutions are priced based on the length videos, CloudVideo is priced based on the underlying VM usage. CloudVideo is able to automatically transcode jobs submitted by users by launching and terminating VMs according to different policies. Reliability is ensured by gracefully handling failures of slaves. The system supports overlapping of the CPU-intensive and network-intensive parts of jobs, resulting in higher utilization of resources. When evaluating CloudVideo on realistic workloads we compare the performance of three different policies for allocating VMs and find that a policy that aims to launch new VMs ahead of demand ends up using the least number of total VMs. We show CloudVideo to be more cost efficient than Amazon Elastic Transcoder while taking longer time to complete jobs, making CloudVideo the optimal choice for cost aware organizations.**

I. Introduction

Transcoding of videos, that is taking a video file and converting it to another format, has traditionally been done on stand-alone machines. With the emergence of cost efficient cloud based systems the opportunity of moving transcoding of videos to the cloud proves to be interesting.

Existing solutions such as Zencoder [1] and Amazon Elastic Transcoder [2] provide the option of transcoding in the cloud. These solutions both price the service based on the length of the transcoded video files. We want to explore the possibilities of using a custom-made cloud based transcoder, which we name CloudVideo.

While the previously mentioned providers price transcoding based on video length, we present a solution where the pricing of the underlying VMs is the deciding factor and explore the difference this pricing model results in. Additionally, we will explore different policies for launching and terminating VMs as well as integrate caching into CloudVideo to avoid having to do the same transcodings multiple times.

The organization of the paper is as follows: Section II provides a background on the application and its requirements. Section III describes the design of CloudVideo and the policies it supports. We present the setup of experiments along with results from experiments in Section IV. In Section V we summarize our main findings and discuss trade-offs, and conclude in Section VI.

II. Background on Application

The goal of CloudVideo is to be able to elastically transcode videos in the cloud. A use-case could be to transcode videos uploaded from users to a variety of formats, as needed by videos sharing sites such as Youtube. The actual transcoding of the videos is done by using FFmpeg [3]. Since FFmpeg is able to utilize multiple cores on a single machine, we have decided to have jobs be executions of FFmpeg which are run on single VMs.

A. Requirements

The following key requirements are fulfilled by our system:

1) Automation: The system works autonomously after a user has submitted a job, handling the features mentioned below without human intervention.

2) Elasticity: New VMs are launched and terminated according to the demand. Multiple different policies for evaluating when to launch and terminate VMs are supported.

3) Performance: Jobs are allocated to VMs as soon as possible. To further improve performance overlapping of jobs is possible.

4) Reliability: The system is able to handle failing jobs and slave VMs that are lost.

5) Monitoring: The resource usage is monitorable such that we are able to conduct experiments.

6) Caching: If the same transcoding job is submitted multiple times the actual transcoding is only done once.

III. System Design

A. Using CloudVideo as a user

Before diving into the structure of the design of CloudVideo we will consider the workflow of transcoding videos. Since the automation requirement demands minimal interaction with the users, the process of transcoding a video is simple for users. A figure of the process of transcoding using CloudVideo from the perspective of a user can be seen in fig. 1 on the following page. Using our client, the user simply picks a video to be transcoded and determines the output format. This video is then uploaded to S3 from where CloudVideo automatically transcodes the video and places the output file back on S3 for the user to download.

B. Resource Management Architecture

The three main components of the system are: A Master, Slave and Client. These components and the interoperability between them can be seen in fig. 2 on the next page. In

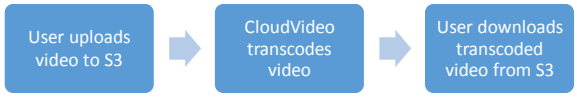


Fig. 1: The process of using CloudVideo as seen from the perspective of a user

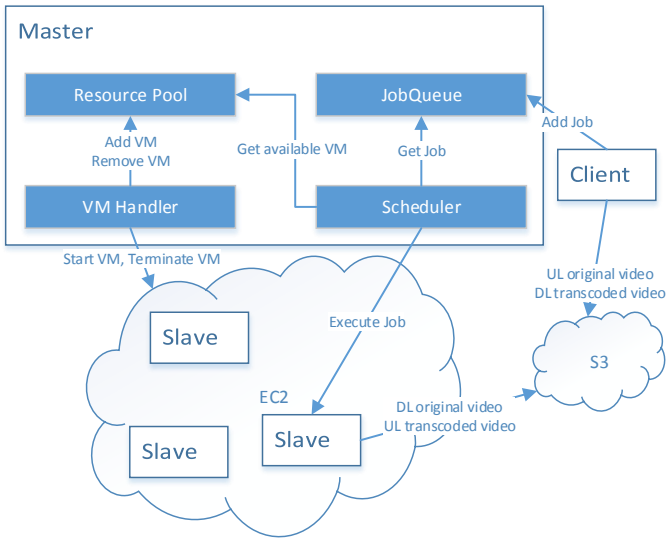


Fig. 2: The three main components of CloudVideo (Master, Slave, Client) and their interoperability

this section we will first briefly describe the design of the Client and Slave after which the working of the Master will be described in detail.

1) Client: The main role of the client is to be used by end-users. Its functionality is simply to first upload a video to S3, then submit a request for transcoding to another format to the server and lastly to wait for notification that the job has completed. All communication with the server is done using TCP connections.

2) Slave: When slaves are started, they listen for connections from the master. We use Java RMI to communicate with slaves. When connected, the master is able to execute jobs on the slaves. Once a job is started on a slave, it is marked as running. Once a job has completed, the slave is marked as not running, after which the master can submit another job. The slave also has a kill switch which can be used to simulate failures.

The process of actually running a job on the slave consists of the following steps:

- The slave immediately starts FFmpeg, which fetches the input file directly from S3 using HTTP (which is faster than first downloading the file locally and then transcoding it).
- After finishing executing FFmpeg the slave then uploads the transcoded file to S3.

3) Master: The main logic of the system is defined in the master. Which consists of the following components:

a) Resource Pool: The responsibility of the resource pool is to keep a collection of slaves. It is able to add and remove slaves from the pool in a thread safe manner as well as return a slave that is available for running a job. In case a slave in the pool is found to not reply to connections, the resource pool notifies the VM Handler to take care of the issue.

b) Job queue: The goal of the job queue is to hold the jobs in the system that are waiting to be executed. When the scheduler requests a job from the queue it is returned according to the queue policy that is used.

c) VM Handler: The VM Handler supports different policies that, by looking at the amount of historical jobs or the amount of jobs in the queue, determine the needed number of VMs. It is then able to start and terminate VMs in the cloud (in our case EC2), thus fulfilling the elasticity requirement. Likewise, if other components detect that a slave is not running properly, the VM Handler is able to terminate the slave and launch a replacement, thus ensuring reliability.

The actual VMs are launched using an Amazon Machine Image (AMI) which on startup runs a script that downloads and executes the newest version of the slave from S3. The VM Handler repeatedly polls the VM until the slave is running, after which the VM can be added to the resource pool.

d) Scheduler: The Scheduler uses information from the other components to get the next job that should be executed from the queue and subsequently getting an available slave for running the job on. The actual execution of a job is monitored by a thread, such that the job can be relaunched on another slave in case the slave crashes. The scheduler is thus the component responsible of satisfying the performance requirement.

e) Monitoring: To satisfy the requirement of monitoring, we have added various logging capabilities to the components in the system. Thus the data provided for use in the experiments is gathered from different components:

- The client records the total time, from the time a job was submitted until its completion.
- Slaves log their total resource consumption (CPU, Memory, Network, Disk I/O).
- The master at any given time logs the amount of slaves running, the amount of slaves currently being launched, the number of jobs in the queue, and the number of jobs running.
- Additionally both clients, slaves and the master keep a verbose log-file of any event happening such that one can later see exactly what happened in the system.

C. System Policies

1) Scheduling policy: The job queue is currently a FIFO queue for determining the order in which to run jobs. Other policies, such as prioritizing users differently could easily be implemented by using a priority queue instead.

In case a job or slave has failed, the job is inserted at the front of the queue to prioritize retrying the job.

2) Job overlap policies: CloudVideo supports two different policies for executing jobs.

- The next job is only started if the previous job has completely finished.
- The next job is started as soon as the previous job is done transcoding, but before it has finished uploading the transcoded file.

3) VM Handler policies: Since scalability is an important feature of CloudVideo, the VM Handler currently supports three different policies for scaling. As exactly one slave is run on each VM and at most one job on each slave, scaling in CloudVideo means starting new VMs and terminating already running VMs.

a) Policy 1: Percentage rule policy: This policy is based on the idea that clients should be served as quickly as possible, but we also should be careful starting up VMs because of the time it costs to start them. To accomplish this, we compare the jobs in the queue with the slaves we have, plus a small percentage on top of that. Algorithm 1 shows the the pseudo-code for this policy. With pending, we mean the amount of pending VMs (slave isn't running yet). With average, we mean the average amount of jobs in the queue over the last minute. With available, we mean the amount of slaves ready to run a job.

if *average >1.1 * (available + pending)* **then**
  launch a VM;
end
if *average equals 0* **then**
  terminate a VM;
end

Algorithm 1: Percentage rule policy

b) Policy 2: Job Queue Policy: This policy is is based on the idea that simpler is better (note that we do not mean to state that simpler is better). Simply starting up VMs if there are jobs in the queue is one of the simplest way of scaling. Algorithm 2 shows the the pseudo-code for this policy. With jobs, we mean the amount of jobs currently in the queue.

if *jobs >pending* **then**
  launch a VM;
end
if *average equals 0* **then**
  terminate a VM;
end

Algorithm 2: Job Queue Policy

c) Policy 3: Available Slaves Policy: Like the Percentage Rule Policy, this policy is also based on the idea that we should serve clients as fast as possible. To achieve this, this policy tries to stay ahead of the curve, by already starting new VMs before all slaves are running a job. Algorithm 3 shows the the pseudo-code for this policy.

D. Additional System Features

1) Caching: Since the same file might be submitted for transcoding multiple times, we want to avoid having to do the exact same transcoding multiple times. This is handled in the following way: When users upload a file to S3, its hash (fetched directly from S3 without having to download the file) is compared with the hashes of files previously uploaded in the system. In case the hashes match, and the requested output format is also the same, we simply return the location of the output file from the previous job. This assumes that the output file from the previous job still exists and results in not having to do the exact same transcoding again.

if *available + pending <0.3 * total* **then**
  launch a VM;
end
if *available >0.5 * total* **then**
  terminate a VM;
end

Algorithm 3: Available Slaves Policy

File name	File size
anchorman2-tlr2b_h480p.mov	20.3 MB
captainamericathewintersoldier-tsrtlr1_h480p.mov	19.0 MB
The.Avengers.Trailer.1080p.mov	24.5 MB
allwifedout-tlr_h480p.mov	14.6 MB
charliecountryman-tlr_h480p.mov	27.4 MB
nonstop-tlr1_h480p.mov	18.6 MB
philomena-tlr1r_h480p.mov	25.2 MB
thatakwardmoment-tlr1_h480p.mov	19.3 MB

Table I: Movie files

IV. Experimental Results

In this section the experimental results of CloudVideo are presented.

A. Experimental setup

The slaves run on VMs on Amazon EC2. We use the cheapest instance, which is a t1.micro instance. VMs run a modified Amazon Linux AMI 2013.09 (ami-149f7863 (64-bit)) image on the instance, which starts up a slave instance after the OS has completed booting. We used the *EU (Ireland)* region for both the S3 bucket and EC2.

The Master runs on a Intel Core i7-3610QM (quadcore, 2.3GHz per core, with hyperthreading enabled) machine, with 8 GB of RAM and is running Windows 7 (64-bit).

To run experiments on CloudVideo we created a client simulator that starts up clients that connect to the master (via the client server) and request the transcoding of a certain video file. For all our test we use .mov files, and transcode them to .mp4 files. In table I you can see the files we used for the experiments. Since we can only run a maximum of 17 VMs on EC2 and only use the cheapest instances (with the accounts and resources we had at our disposal), we concentrated on relatively small workloads in our experiments.

We used two external libraries for CloudVideo. The first one is the AWS SDK for java [4], provided by Amazon, to be able to use EC2 and S3. The second library is the Uncommons Maths library [5], which we used to generate workloads. To monitor resource usage on slaves we used the `sar` command from the `sysstat` package [6].

Normally a client first has to upload a file to S3, so that one of the slaves can transcode the file. Since uploading to S3 is handled by the clients themselves and the CloudVideo master has no influence on the transfer rates between the client and S3, we disabled that feature, and pre-load all the movie files on S3. This way, we save some time when running experiments. Note that the slaves do upload the transcoded files to S3, which we will consider in section IV-C. Except for the experiment on caching, the caching feature was disabled for all tests.

B. Experiment: Measuring overhead

To measure the actual overhead of running jobs, we submit a series of very small files for conversion in succession. Since FFmpeg also supports conversion of images, we decided to convert small images. In practice, we submit a 152 byte .png image which should be converted to a .tiff image. This is a sufficiently short-running operation to be able to measure the time it takes from a job is submitted until it returns. Thus we can measure the overhead of putting a job into the queue, removing a job from the queue, finding a slave from the resource pool to execute the job on, actually executing the job, and finally returning the result to the client.

For the experiment we ran a single slave in the cloud and a remote master on a laptop. On the same laptop, a single client was launched, which submitted a job of converting a 152 byte image. Caching on the master was disabled to force the master to actually submit all jobs to the slaves. The time from the client was launched until it received the file back was measured as the overhead. The experiment was repeated 1000 times and resulted in an average overhead of 258 ms with a standard deviation of 82 ms. We find that this low overhead is satisfying, since it represents the total time from a user submits a job until its completion.

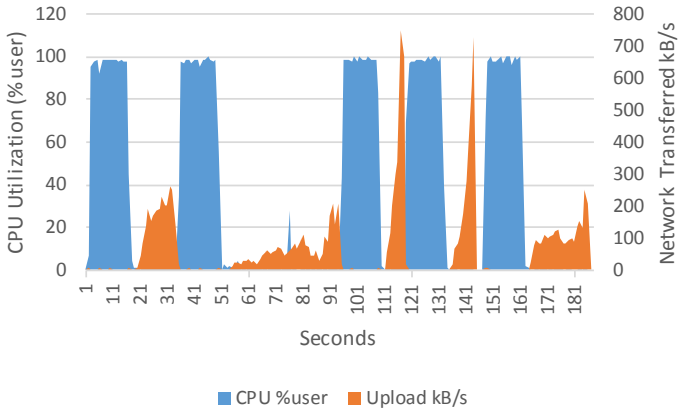
C. Experiment: Overlapping jobs

We observe that jobs stress different resources during their execution: First jobs are CPU-intensive while FFmpeg is running. Then, jobs are network-intensive while uploading the resulting file to S3. In our normal setup, slaves are running on EC2 in the same region as the S3 buckets they are utilizing. This results in high transfer rates between VMs and S3, which means the limiting factor in general is CPU.

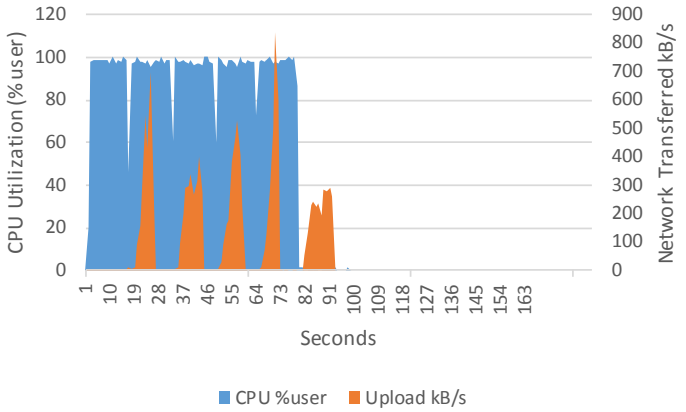
To consider another setup, where not all slaves are located near the storage they upload the final output to, we utilized a slave running in the *EU (Ireland)* region which uploaded files to an S3 bucket in *Asia Pacific (Tokyo)*. We run a test where 5 identical jobs are queued for execution immediately after each other. Caching is disabled to ensure transcoding is done for each job.

The result of running these 5 jobs without overlapping computation and network transfers can be seen in fig. 3(a). As it can be seen from the figure, transfer speeds between the Ireland and Tokyo region vary considerably and incur a significant overhead.

In comparison, fig. 3(b) shows the same jobs running on the same machine, where the slaves are marked as available for running a new job as soon as the transcoding has finished. As a result it can be seen that the the utilization of computation



(a) No overlap between uploads and computation



(b) Overlapping the upload of one job with the computation of a following job

Fig. 3: Executing 5 jobs back-to-back with and without overlapping uploads with computation

resources becomes much higher, making it worthwhile to overlap computation with transfer.

D. Experiment: Realistic workloads

We used a pseudo-random generator, using a exponential distribution, to generate different workloads to test CloudVideo. The workloads differ in the amount of job requests per minute and the total amount of job requests the Client simulator generates. We ran these workloads on CloudVideo for each of the three VM Handler policies. Furthermore, we also ran experiments with different minimum amounts of VMs (i.e. the VM Handler can scale the amount of VMs up, but not scale down further than the minimum amount of VMs). The maximum amount of slaves we use is 17 for all tests.

To get a feeling for how the scaling policies work, one can consult Fig. 4. The figure shows the number of pending/running VMs, the number of jobs in the queue and the number of slaves running a job, during a Client simulator run (the Policy 2 run of table IV). We can see that Policy 2 reacts quite quickly to a rising job queue, and starts new VMs almost instantly. It also quickly terminates VMs after slaves finish their job and there are no more jobs in the queue.

Table II, III and IV show the results of the different workloads. The times shown are in seconds. The total time is

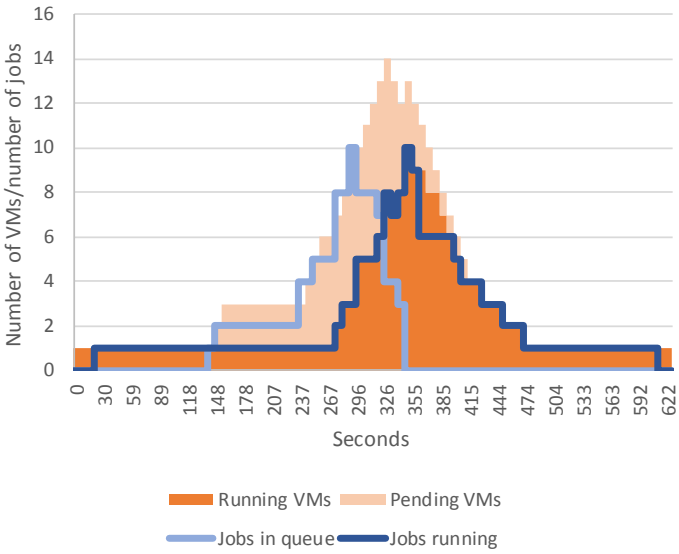


Fig. 4: The provisioning of VMs done by policy 2 compared to the amount of jobs. TODO: Figure text + write about me

	Total time	Average time	Total VMs	Cost (€)
Policy 1	522	294	4	0.40
Policy 2	637	331	6	0.60
Policy 3	466	241	6	0.60
Minimum 5	452	184	5	0.50

Table II: Results for workload of 1 job per minute, 5 jobs in total. Time is in seconds.

the time between the starting time of the Client simulator and the time all jobs finished. The average time, is the average of the time each client had to wait before its file was transcoded. The policies correspond to the policies previously mentioned, and were all executed with a minimum amount of VMs of 1. The minimum 5, 10 and 15 are the results of executing with a minimum amount of VMs of 5, 10 and 15 respectively (all used policy 1). Since all these workloads finish within the hour, the total VMs reported, are equal to the time that would have been charged using the Amazon EC2 timing approach. The cost column shows the costs associated with the VM usage, assuming €0.10 per VM-hour.

Table II shows the results for running a workload of 1 job per minute and 5 jobs in total on CloudVideo. For a small amount of jobs, policy 2 and 3 start up more VMs than they need. Policy 1 seems a bit more conservative, but has a lower total run time than policy 2 nonetheless. Policy 3 comes close to the performance of minimum 5, probably because it already starts up new VMs before the pool of available slaves runs out.

Table III shows the results for running a workload of 2 jobs per minute and 10 jobs in total. Although this workload is only a little higher than the previous, the 3 policies struggle to come close to the performance of minimum 10. This partly has to do with the fact that starting a new VM takes at least a minute, so even if 1 of the policies would immediately start 9 new VMs, they would still lag a little behind minimum 10. More surprising is that policy 1 uses the most VMs, but takes the longest time to finish. This is probably because it too aggressively terminates VMs, after which it starts up a new

	Total time	Average time	Total VMs	Cost (€)
Policy 1	634	363	14	1.40
Policy 2	546	304	11	1.10
Policy 3	597	329	8	0.80
Minimum 5	477	191	5	0.50
Minimum 10	397	172	10	1.00

Table III: Results for workload of 2 jobs per minute, 10 jobs in total. Time is in seconds.

	Total time	Average time	Total VMs	Cost (€)
Policy 1	692	392	15	1.50
Policy 2	622	332	15	1.50
Policy 3	767	442	11	1.10
Minimum 5	547	254	10	1.00
Minimum 10	437	183	10	1.00
Minimum 15	413	179	15	1.50

Table IV: Results for workload of 3 jobs per minute, 15 jobs in total. Time is in seconds.

VM again. Policy 2 is the fasters of the 3 policies, but uses a little bit more VMs than policy 3.

Table IV shows the results for running a workload of 3 jobs per minute and 15 jobs in total. We can see that Policy 2, again, has the lowest total and average time of the three policies. But we can also see, that policy 3 is the most economical in terms of VMs used. And, again, the policies struggle to come close to the performance of using a minimum amount of VMs of 10 and 15.

E. Experiment: Caching

To ensure that caching is working as expected we first ran the workload consisting of 1 job per minute and 5 jobs in total. After successfully completing, the exact same workload was submitted again. To ensure that caching actually worked, we manually verified that the returned files were the expected videos and correct formats.

While the average time from submission to completion for a job in the first run was 5.5 minutes, the average time to completion in the second run was 3 seconds. Additionally, no extra slaves were launched in the second run, since all requests could be served directly form the cache. This confirms that caching works as expected.

V. Discussion

A. Comparison with Amazon Elastic Transcoder

Instead of only extrapolating our results on their own, we have chosen to also compare our solution with an existing product in the same market as ours: Amazon Elastic Transcoder [2]. While the cost of CloudVideo is related to the amounts of VMs used, the cost of using Amazon Elastic Transcoder depends on the length of the video. Note that we compare the price and costs in dollars instead of euros, because Amazon gives its prices in dollars. We also took the up-to-date prices of EC2, to be able to make a fair comparison with the elastic transcoder.

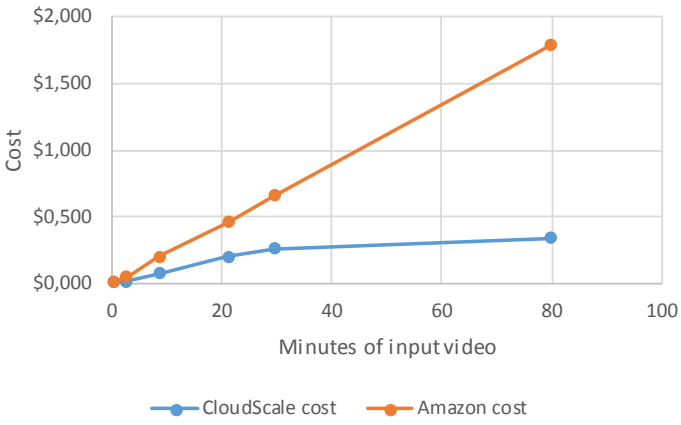


Fig. 5: Cost compared to the length of input videos for Amazon Elastic Transcoder and CloudVideo

	Price
Amazon Elastic Transcoder 1 minute of SD video	$0.017
Amazon Elastic Transcoder 1 minute of HD video	$0.034
CloudVideo 1 hour of computation (EC2 Micro)	$0.020

Table V: Base prices of Amazon Elastic Transcoder versus CloudVideo

1) Costs: A comparison of the costs of Amazon's solution versus CloudVideo as a function of the duration of the input videos can be seen in fig. 5. In generating the graph the input files and job arrival times described in Section IV-D are used. As it can be seen from the graph, Amazon's pricing scales linearly (actually almost linearly, since there is a single HD file in the workload which is billed at a higher cost). CloudVideo, on the other hand, can be seen to benefit from economies of scale: Once a VM is launched it is billed for the whole hour.

Looking at the prices in table V makes it clear that CloudVideo easily is able to hold an advantage over Amazon Elastic Transcoder: For the price of converting a single minute of video a CloudVideo slave can be kept running for 1 hour.

To further quantify the difference in performance, we found our slaves able to transcode the file `AM_RedBand_DomTrailer1_720p.mov` in 33.5 minutes. Taking the length of the video into account, this translates to a single CloudVideo slave being able to transcode 4.8 minutes of HD video per hour at a cost of $0.020 (which would cost $4.8 \cdot \$0.034 = \$0.164$ using Amazon Elastic Transcoder, representing a $\frac{0.164}{0.020} \approx 8$ times more expensive solution assuming full load).

2) Processing time: Price-wise there are thus great gains to get from utilizing CloudVideo. Time is another matter, though. While Amazon Elastic Transcoder was able to convert the above mentioned file in an average time of 1.6 minutes from submission until completion, CloudVideo, as mentioned, uses 33.5 minutes for the same file, assuming no other jobs are waiting in the queue. Thus the decision of whether to use Amazon Elastic Transcoder or CloudVideo can be stated as a trade off between cost and job processing time (we have assumed that CloudVideo only uses EC2 Micro instances. The picture will be different with more expensive instances with more processing power).

B. Extrapolation of results

From our results in section IV, it seems that Policy 3 is the most economic, in terms of VM usage. If we were to extrapolate the results from table IV, for example for 100,000/1,000,000/10,000,000 users, we would get €7,333/€73,330/€733,300 respectively. And, if we were to extrapolate the results from table IV, for example for 1 day/1 month/1 year, we would get €124/€3,841/€45,228 respectively. In reality, the real life costs would presumably be lower than this, since our current experiments do not run for a full hour.

C. Amazon pricing

The policies CloudVideo uses for scaling the VMs up and down, do not take into account the 1-hour increment pricing of Amazon. However, we think CloudVideo could be altered in such a way, that this is taken into account. For example, we could add a policy that decides on the basis of the amount of jobs in the queue, in combination with the average running time per job, whether a new VM should be started or not.

VI. Conclusion

We have presented CloudVideo, a solution for transcoding videos in the cloud. CloudVideo operates automatically and has the option of using different policies for allocating VMs. We showed that a policy for launching VMs before all existing VMs are used to be the most efficient in terms of the number of VMs used.

When submitting tiny transcoding jobs we found that the time from submission to completion is 258 ms, representing the overhead in the system. A comparison of running jobs back-to-back while overlapping computation with network transfer was presented. In cases where network transfer rates are low this approach of overlapping results in a significant performance increase.

CloudVideo supports caching, such that exact same transcodings are not re-done in case a similar job is later submitted. Caching was confirmed to work in an experiment where average job times were reduced from 5.5 minutes to 3 seconds.

In a comparison with Amazon Elastic Transcoder we showed CloudVideo to be more cost efficient, with Amazon Elastic Transcoder being up to 8 times more expensive compared to CloudVideo under full load. On the other hand, times from job submission until completion were lower in Amazon Elastic Transcoder, representing a trade-off between cost and processing time.

Appendix A
Source code

The source code for this project is publicly available at: https://github.com/henrikhannemose/CloudVideo.

APPENDIX B
TIME SHEETS

A. Time for experiments

	total-time	dev-time	setup-time
Measuring overhead	2	1	1
Overlapping jobs	3	1	1
Realistic workloads	8	4	1
Caching	2	1	1

B. Total time

total-time	112
think-time	10
dev-time	55
xp-time	15
analysis-time	10
write-time	12
wasted-time	0

REFERENCES

[1] "Zencoder," http://zencoder.com/.

[2] "Amazon elastic transcoder (beta)," http://aws.amazon.com/elastictranscoder/.

[3] "Ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video," http://www.ffmpeg.org/.

[4] "Aws sdk for java, version 1.6.1," http://aws.amazon.com/sdkforjava/.

[5] "Uncommons maths library, version 1.2.3," http://maths.uncommons.org/.

[6] "Sysstat utilities," http://sebastien.godard.pagesperso-orange.fr/.