CloudStream: delivering high-quality streaming videos through a cloud-based SVC proxy

Zixia Huang[1], Chao Mei[1], Li Erran Li[2], Thomas Woo[2]
[1]Department of Computer Science, University of Illinois, Urbana, IL
[2]Bell Labs, Alcatel-Lucent, Murray Hill, NJ
[1]{zhuang21, chaomei2}@illinois.edu
[2]{erranlli, woo}@research.bell-labs.com

***Abstract*—Existing media providers such as YouTube and Hulu deliver videos by turning it into a progressive download. This can result in frequent video freezes under varying network dynamics. In this paper, we present CloudStream: a cloud-based video proxy that can deliver high-quality streaming videos by transcoding the original video in real time to a scalable codec which allows streaming adaptation to network dynamics. The key is a multi-level transcoding parallelization framework with two mapping options (*Hallsh-based Mapping* and *Lateness-first Mapping*) that optimize transcoding speed and reduce the *transcoding jitters* while preserving the encoded video quality. We evaluate the performance of CloudStream on our campus cloud testbed.**

I. Introduction

Users are demanding uninterrupted delivery of increasingly higher quality videos (e.g., 720p) over the Internet, in both wireline and wireless. Instead of tackling the video delivery problem head on, most current Internet media providers (like YouTube or Hulu) have taken the easy way out and changed the problem to that of a progressive download via a content distribution network. In such a framework, they are using a non-adaptive codec, but ultimately, the delivery variabilities is handled by *freezing*, which significantly degrades the user experience. In this paper, we propose and study the development of a H.264/SVC (Scalable Video Coding [1]) based video proxy situated between the users and media servers, that can adapt to changing network conditions using scalable layers at different data rates. The two major functions of this proxy are: (1) video transcoding from original formats to SVC, and (2) video streaming to different users under Internet dynamics.

Because of codec incompatibilities, a video proxy will have to decode an original video into an intermediate format and re-encode it to SVC. While the video decoding overhead is negligible, the encoding process is highly complex that the transcoding speed is relatively slow even on a modern multi-core processor. This results in a long duration before a user can access the transcoded video (called *video access time*), and possible *video freezes* during its playback because of the unavailability of transcoded video data. Both long access time and frequent freezes directly and negatively impact the users' subjective perceptions of the video. To enable real-time transcoding and allow scalable support for multiple concurrent videos, our video proxy employs a cluster of computers or a cloud for its operation. Specifically, our proxy solution partitions a video into clips and maps them to different compute nodes (instances) configured with one or multiple CPUs in order to achieve encoding parallelization. In general, video clips with the same duration can demand different computation time (Fig. 1) because of the video content heterogeneity. In performing encoding parallelization in the cloud, there are three main issues to consider. First, multiple video clips can be mapped to compute nodes at different time (Map time) due to the availability of the cloud computation resources and the heterogeneity in the computation overhead of previous clips (Fig. 2). Second, the default first-task first-serve scheme in the cloud can introduce unbalanced computation load on different nodes. This will lead to the deviations from the expected arrival time at the Reduce application (the encoding completion time or the Reduce time) of different video clips (Fig. 2). The deviation is called the *transcoding jitter*. Third, the transcoding component should not speed up video encoding at the expense of degrading the encoded video quality.

Due to the transcoding jitters and the resulting mandatory reordering buffer at the Reduce application, the SVC-encoded video clips can arrive at the streaming component in batches (Fig. 2). This complicates the streaming component in that the batched arrivals can create a demand surge of network resources due to a sudden data rate increase. Hence, some data may not arrive at the user before the scheduled playback time because of the Internet bandwidth variations and streaming adaptations, which again can result in video freezes. We use the term *streaming jitter* to describe the deviation from the expected arrival time of a video clip at the user.

The video access time and video freezes are user-observable attributes characterizing the *streaming quality*. The video freezes can be reduced by increasing the buffering time at a user. This, however, will in turn increase the access time because the buffer size is decided before the start of an on-demand video [2]. Hence, the minimization of jitters incurred at both transcoding and streaming components is important in improving the overall streaming quality.

Our contributions. We present CloudStream: a cloud-based SVC proxy that delivers high-quality Internet streaming videos. We focus on its transcoding component design in this paper. To achieve this, we characterize the video content

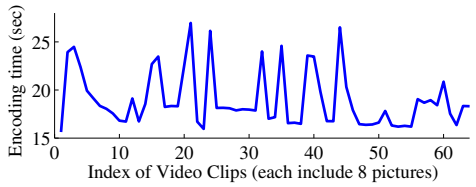

Fig. 1. Encoding time of 64 consecutive video clips (each including eight 856x480 pictures) on a single-thread compute node.

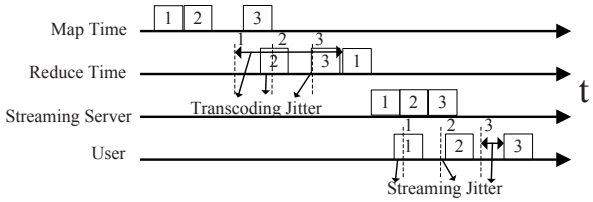

Fig. 2. Transcoding and streaming jitters. Dotted lines show the expected arrival time for each video clip at different components.

heterogeneity and identify the user-observable and system-controllable attributes impacting the video streaming quality. We design a multi-level parallelization framework under the computation resource constraints with two mapping options (*Hallsh-based Mapping* and *Lateness-first Mapping*) to increase the transcoding speed and reduce the *transcoding jitters* without losing the encoded video quality. We evaluate the performance of CloudStream on our campus cloud testbed. Due to the space limit, the video streaming adaptation component is deferred to a future full-version paper, and we assume the streaming jitters are equal to transcoding jitters for each clip in this paper.

Because we use a campus cloud testbed, the communication latency between the video proxy and the testbed is neglected for simplicity. We assume all the compute nodes in the cloud have the same computation power. We partition a video into multiple clips with the same duration (number of pictures). We use the SVC reference software JSVM in evaluation, but our system can be migrated to other video codecs.

II. Metrics Affecting Streaming Quality

A. Attributes of Streaming Quality

The streaming quality is a prerequisite for users to watch videos smoothly without interruptions, and thus directly impacts the human subjective perception.

The access time that a user experiences before the start of an on-demand video playback represents the overall responsiveness of the video proxy. The latencies incurred at both transcoding and streaming components can contribute to the access time. In this paper we focus on the deterministic system-controllable factor, and specifically minimizing the average latency spent over encoding one video clip on the cloud compute node, because a video cannot be accessed until one or multiple re-encoded video clips have been returned from the cloud and arrived at the user.

Video freezes are caused by the unavailability of new video data at their scheduled playback time due to the combined contribution of transcoding and streaming jitters. For the i-th video clip, we use c_i to denote the Reduce time (the actual completion time). Each video clip has a encoding time p_i. In order to enable real-time transcoding, the expected encoding completion time (Reduce time) of the video clips d_i is defined as $d_i = p_e + (i-1) \times \Delta T$ where p_e is the expected encoding time of a video clip (a constant) and ΔT is the duration of the video clip in time. The term $(i-1) \times \Delta T$ computes the temporal shift of the i-th video clip from the start of the video. The transcoding jitter of each clip δT_i^t can therefore formally defined as: $\delta T_i^t = c_i - d_i$. In this paper, we assume the streaming jitter δT_i^s is equal to the transcoding jitter for each clip, i.e., $\delta T_i^s = \delta T_i^t$. The user-side buffering time should be large enough to accommodate the maximum streaming jitter in order to avoid video freezes. The video decoding time is negligible at both the transcoding component and the user.

B. Metrics Characterizing Video Content

The characteristics of the video content can affect the transcoding speed which decides the streaming quality. We use two cost-effective metrics in ITU-T P.910 [3] to describe the video content heterogeneity: the temporal motion metric TM and the spatial detail metric SD. TM can be captured by the differences of pixels at the same spatial location of two pictures, while SD is computed from the differences of all spatially neighboring pixels within a picture. Details can be found in [3], and will not be presented here.

Generally speaking, frequent scenery changes (large TM) affect the computation demand on video encoding by reducing the dependencies among the temporal successive pictures and increasing the number of pictures being intra-coded [1] in response to a scenery change. Pictures with large SD can also increase the encoding computation overhead because of a demand for greater information to describe the spatial details [1].

III. Cloud-Based Video Transcoding Framework

A. Multi-level Encoding Parallelization

We propose our parallelization framework. Previous studies have achieved the SVC encoding parallelization on a single multi-core computer with and without GPU support [4], but none of them provide the real-time support.

SVC coding structures. SVC [1] can divide a video into non-overlapping coding-independent groups of pictures (GOP). Each picture may include several layers with different spatial resolutions and encoding qualities [1]. Each layer in a picture can be divided into one or more coding-independent slices. Each slice includes a sequence of non-overlapping 16x16 macro-blocks (MB) for luma components and 8x8 MB for chroma components. There are strong interdependencies among MBs within a slice [4]. To achieve the best quality inside a MB, the SVC encoder in JSVM compares all independently computed partitions within a MB. Hence, the parallelism at all levels from GOPs (the largest units) to MB partitions (the smallest units) can lead to a corresponding decrease in the work granularity.

Inter-node and intra-node parallelism. To exploit the compute nodes returned by the cloud at full potential, we need to parallelize the encoding scheme across different compute nodes (*inter-node parallelism*) which do not share the memory

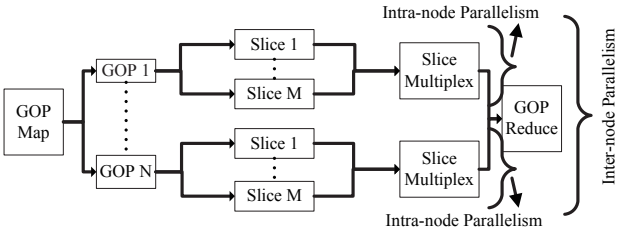

Fig. 3. Multi-level parallelization in the cloud.

space. On the other hand, the shared-memory address space of the parallelism inside one compute node (*intra-node parallelism*) will ease the management of sharing information or states with the help of threads synchronization by locks or barriers. However, threads synchronization causes overhead and may serialize the whole encoding, offsetting the performance gains from parallelization. Additionally, these two different types of parallelism prefer different work granularity. As a rule of thumb, the inter-node parallelism has much larger granularity than its counterpart.

Choice of our parallelization scheme. From the above discussions above, we propose our multi-level encoding parallelization scheme. The idea of our scheme is shown in Fig. 3. In our scheme, since the coding-independent GOPs have the largest work granularity, encoding at the GOP level is an ideal candidate to be parallelized across different compute nodes. The independence among all slices and the relative larger amount of work makes it ideal for intra-node parallelism, which can be easily implemented using OpenMP by encoding each slice on a different CPU within a compute node (Fig. 3).

B. Intra-node Parallelism

Because a cloud node will not return part of the encoded SVC data until the whole GOP is encoded, the inter-node parallelism alone is not sufficient to shorten the access time. To set an upper bound on the average computation time spent over the GOP encoding that decides the access time, we rely on the intra-node parallelism and decide the minimum number of slices encoded in parallel.

We limit $\mathrm{T}_{\mathrm{GOP}}(M)$ (using M parallel slices) within the upper bound T_{th}. Each GOP includes SG pictures (each with L spatial and quality layers in JSVM configuration), i.e., $\mathrm{T}_{\mathrm{pic}}(M) \times \mathrm{SG} < \mathrm{T}_{\mathrm{th}}$, where $\mathrm{T}_{\mathrm{pic}}(M) = \Sigma_{i=1}^{L} \mathrm{T}_{\mathrm{pic},i}(M)$. $\mathrm{T}_{\mathrm{pic},i}(M)$ can be computed in approximation by:

$$\mathrm{T}_{\mathrm{pic},i}(M) = \mathrm{T}_{\mathrm{slice},i}(M) \times \mathrm{N}_{\mathrm{slice},i}/M + \Delta \mathrm{T}_i \quad (1)$$

$\Delta \mathrm{T}_i$ is the encoding overhead which cannot be parallelized. It includes the preprocessing overhead of the layer i and the multiplexing overhead of the encoded slices belonging to the same layer of a picture. $\mathrm{T}_{\mathrm{slice},i}(M)$ can be computed by:

$$\mathrm{T}_{\mathrm{slice},i}(M) = \mathrm{T}_{\mathrm{MB},i}(M) \times \mathrm{N}_{\mathrm{MB},i}/\mathrm{N}_{\mathrm{slice},i} \quad (2)$$

Hence given SG and $\mathrm{T}_{\mathrm{MB},i}(M) = \mathrm{T}_{\mathrm{MB},i}(1)$ (because we do not parallelize across or within MBs in a slice), the minimum M that can satisfy T_{th} is:

$$M_{\min} = \left\lceil \frac{\sum_{i=1}^{L} \mathrm{T}_{\mathrm{MB},i}(1) \times \mathrm{N}_{\mathrm{MB},i}}{\mathrm{T}_{\mathrm{th}}/\mathrm{SG} - \sum_{i=1}^{L} \Delta \mathrm{T}_i} \right\rceil \quad (3)$$

The numerator of Eq. 3 indicates the sequential encoding time of all MBs at all layers of one picture, while the denominator is the upper bound of the target encoding time for these MBs. Hence, Eq. 3 gives us the minimum requirement of speedup by intra-node parallelization given the upper bound T_{th}, which correspondingly implies the minimum number of slices (i.e. number of intra-node parallelism) which should be used in our scheme. Note that $\mathrm{T}_{\mathrm{MB},i}$ and $\Delta \mathrm{T}_i$ can be obtained based on the encoding time statistics of previous videos in the cloud.

TABLE I
NOTATIONS AND DEFINITIONS

Notations	Definitions
M	Number of encoded parallel slices in a picture
$\mathrm{N}_{\mathrm{MB},i}, \mathrm{N}_{\mathrm{slice},i}$	Number of MBs or slices in the i-th layer of a picture
$\mathrm{T}_{\mathrm{MB},i}(M), \mathrm{T}_{\mathrm{slice},i}(M)$	Average encoding time of one MB or slice in the i-th layer with M parallel slices
$\mathrm{T}_{\mathrm{pic},i}(M)$	Average encoding time of the i-th layer of a picture
$\mathrm{T}_{\mathrm{pic}}(M)$	Average encoding time of a picture
$\mathrm{T}_{\mathrm{GOP}}(M)$	Average encoding time of a GOP

C. Inter-node Parallelism

While the inter-node parallelism is adopted to achieve real-time transcoding, the variations of GOP encoding time can introduce the transcoding jitters. Our goal is to minimize both transcoding jitters and the number of compute nodes (denoted as N) used for inter-node parallelism. The optimization problem requires the estimation of the actual encoding time of the GOPs, which can be achieved by profiling [5], [6]. In our study, we rely on a multi-variable regression model based upon the video content characteristics TM and SD at a given encoding configuration $\mathcal{E} = \{\mathrm{SG}, M, \mathrm{L}, \ldots\}$, and along with the profiling results, to improve the prediction accuracy. We train videos with different TM and SD to build the regression model. Even though there are limited number of videos used for training, we still find that more than 90% of predicted values of the testing data are fallen within the 10% of error, which proves the model is creditable. We will not go into the details due to the space limit.

Based on the approximation of each GOP's encoding time (denoted as $\hat{p}_i$ as opposed to the actual encoding time p_i, because p_i cannot be obtained beforehand), we formulate the following problem. We will still use p_i for illustration simplicity in this section, but it is actually the value of $\hat{p}_i$.

• Problem Formulation

We are given Q jobs (1,2,...,Q). Each job i has a deadline d_i, and a processing time p_i. Multiple machines can do the job in parallel, but each job must be processed without preemption on each machine until its completion. We want to bound the lateness of these jobs. That is, the actual completion time c_i must be smaller than $d_i + \tau$ for each job i. The lateness l_i can be computed $l_i = c_i - d_i$. We would like to find the minimal number of machines N and minimize the lateness τ.

In our system, the jobs correspond to the GOP encoding tasks. l_i is the positive transcoding jitter δT_i^t. Hence τ is the upper bound of both transcoding and streaming jitters.

• Computation Complexity

We show that the problem is NP-hard even if the optimal solution may only require two machines.

Theorem 3.1: The minimum number of the machine scheduling problem (minMS) is NP-hard.

Proof. Our reduction is from the partition problem which asks whether a given multiset of integers can be partitioned into two "halves" that have the same sum. Given an instance of the partition problem, we encode in an instance of our minMS problem. For each integer i in the partition problem, we create a job i. Let the total sum be S. The deadlines for all of these jobs are the same, i.e. $S/2$. We set τ the maximum lateness to zero. It is easy to see that the partition problem has a solution iff the minMS problem outputs two machines with $\tau = 0$.

We present it here only for completeness. We discuss two solutions to this NP-hard problem in this paper.

• Solution I: Hallsh-based Mapping (HM)

We first set an upper bound of τ and then compute the minimal number of N that satisfies the upper bound.

This can be achieved by a O(1) approximation algorithm [7]. However, the algorithm is complex because it involves rounding the solution to a linear programming. In this paper, we present a lightweight algorithm called minMS2approx that makes use of the algorithm in [8] as a blackbox. We refer to the identical machine scheduling algorithm in [8] as Hallsh.

The algorithm minMS2approx runs as follows.

1. We pick $\epsilon = \min_i\{(d_i - p_i)/\tau\}$.

2. We run HallSh algorithm by increasing the number of machines k until the maximum lateness $\max_i\{l_i^{(k)}\}$ among all the Q jobs using k machines satisfies $\max_i\{l_i^{(k)}\} < (1 + \epsilon) \times \tau$. We set the machine number at this point to be K, i.e. $\max_i\{l_i^{(K)}\} < (1 + \epsilon) \times \tau$ and $\max_i\{l_i^{(K-1)}\} \geq (1 + \epsilon) \times \tau$. We flag the machines 1,2,...,K as used.

3. The HallSh algorithm returns the scheduling results of all jobs given K. For a job with $c_i - d_i > \tau$ on a particular machine j, we move it, and along with all future jobs on the same machine K, to a new machine $K + j$. We flag the machine $K + j$ as used. We then compute the new completion time c_i for all jobs on the machine $K + j$.

4. N is computed as the number of used machines.

We claim that the algorithm is a 2-approximation algorithm.

Lemma 3.2: The optimal number of machines N_{opt} for the minMS problem is greater than K.

Proof. K is the first value that satisfies $\max_i\{l_i^{(K)}\} < (1 + \epsilon) \times \tau$ and $\max_i\{l_i^{(K-1)}\} \geq (1 + \epsilon) \times \tau$. We let $\tau_{\text{opt}}^{(K)}$ be the optimal lateness with K identical machines. According to [8], $\max_i\{l_i^{(K-1)}\} < (1 + \epsilon) \times {\tau_{\text{opt}}}^{(K-1)}$. Thus, $\tau < {\tau_{\text{opt}}}^{(K-1)}$. Since the maximum lateness of the optimal solution to the minMS problem (which uses N_{opt} number of machines) is no smaller than τ, this means $N_{\text{opt}} > K - 1$. Because N_{opt} is an integer, $N_{\text{opt}} \geq K$.

Theorem 3.3: The minMS2approx algorithm is a 2 approximation to the minMS problem.

Proof. Let job q be the first job that exceeds the lateness $c_q - d_q > \tau$ on the machine j, and s_q be the job start time on the machine j. Given $s_q > d_q - p_q$ and our choice of $\epsilon = \min_i\{(d_i - p_i)/\tau\}$, we obtain that $s_q > \epsilon \times \tau$. Because the job q will be rescheduled on machine $j+K$, the new start time s'_q will be zero. Its completion time now becomes p_q which is within the τ maximum lateness. For all the other jobs that exceed the maximum lateness and are scheduled after q on the machine j, each of them is moved ahead by $s_q > \epsilon \times \tau$. Since none of them are late by more than $\epsilon \times \tau$, all these jobs will satisfy the maximum lateness τ constraint when scheduled on the machine $j + K$. Thus, our algorithm finds a feasible solution. Because we use at most $2K$ machines, this means $2K \geq N_{\text{opt}}$. With Lemma 3.2, we conclude $K \leq N_{\text{opt}} \leq 2K$.

• Solution II: Lateness-first Mapping (LFM)

We first compute the minimal number of N based on the deadline of each job and design a job scheduling scheme to minimize τ given N compute nodes.

1) Deciding the minimum N: To allow real-time video transcoding, we should satisfy:

$$\mathrm{T_{pic}}(M) \times \mathrm{R} < \mathrm{SG} \times N \tag{4}$$

The left side of Eq. 4 is the number of pictures rendered within the video duration of $\mathrm{T_{pic}}(M)$. For example, we let $\mathrm{T_{pic}}(M) = 3$ sec and the frame rate $\mathrm{R} = 24$ fps. In 3 sec, a total of 72 pictures will be rendered. To allow real-time transcoding, we should use up to N compute nodes for N parallel GOPs so that 72 pictures can be encoded in $\mathrm{T_{pic}}(M)$. If $\mathrm{SG} = 8$, we must use a minimum of $N_{\min} = 9$. We assume the overhead on Map and Reduce applications are negligible compared to the video encoding time.

Hence the minimum N satisfying real-time transcoding is:

$$N_{\min} = \lceil \mathrm{T_{pic}}(M) \times \mathrm{R/SG} \rceil = \lceil \Sigma_{i=1}^{L} \mathrm{T_{pic},}_i(M) \times \mathrm{R/SG} \rceil \tag{5}$$

where $\mathrm{T_{pic},}_i(M)$ can be computed by Eq. 1 and 2.

2) Minimizing τ given N: Given N computed in Eq. 5, we apply the following scheduling algorithm for every N jobs. For the n-th (i = 1,2,...,N) job in every N jobs, we take into account the difference in the job completion time and compute its adjusted processing time $p'_i = p_i - (d_i - d_1)$, where d_1 here is the deadline of the first job within the N jobs. We sort the N jobs by the reverse order of p'_i. We schedule the job with the largest p'_i to the first available compute node, the second largest p'_i to the second available node, etc. Using this scheduling, we can successfully minimize τ. We call it *lateness-first mapping* because the algorithm is aimed at minimizing the lateness.

This simple algorithm is optimal in this case where we have N jobs that we need to assign to N machines. The optimality is established by using a simple exchange argument.

Theorem 3.4: The simple algorithm is optimal in the case of N machines and N job cases.

Proof. We prove by contradiction. Assume we have job i and j such that $p'_i > p'_j$. Let the machine we assign them be k, l. Our algorithm makes sure that the available time of k (i.e., a_k) is earlier than l, i.e. $s_i = a_k < a_l = s_j$. Assume our algorithm is not optimal. We let the job with the maximum lateness in our algorithm be job i (it is easy to extend to the case where there are multple jobs with the same maximum lateness). It must be the case that, in the optimal solution, i is scheduled to l, and there exits a job j that is scheduled to k. Now, we swap these two jobs, and we can reduce the maximum lateness of the optimal.

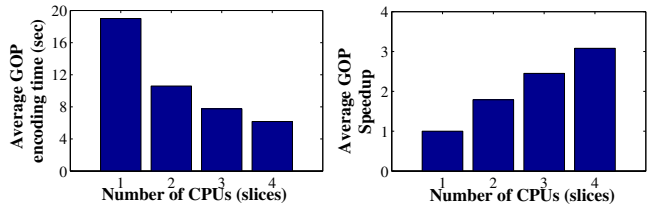

Fig. 4. Average encoding time and speedup using up to 4 cores in intra-node parallelism

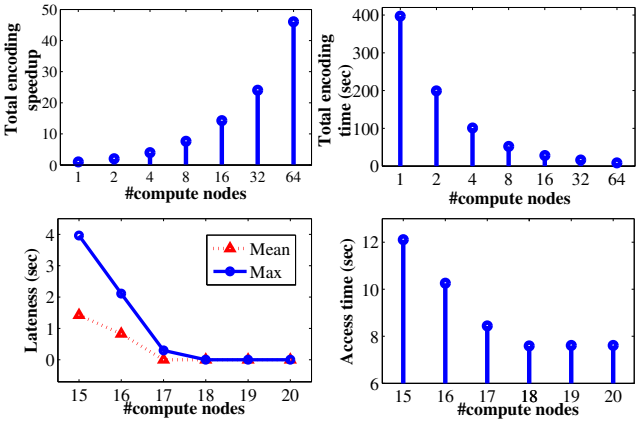

Fig. 5. Lateness-first mapping (LFM) for different number of compute nodes

IV. Performance Evaluation

We evaluate the transcoding performance of CloudStream on the campus cloud testbed. Each compute node in the cluster has up to 8 cores (dual sockets with Intel Xeon E5345 2.33 GHz and 8GB of memory). We set the input as 64 480P video GOPs, each with 8 pictures. Each picture contains 4 temporal layers, 2 spatial layers (856x480 and 428x240) and 1 quality layer. According to our parallelization scheme, each GOP is mapped to an available compute node in the cloud for encoding. We use up to 4 cores on each compute node, and divide each layer to the corresponding number of slices (one core on each slice).

Fig. 4 shows the average encoding time of all the 64 GOPs from 1 core to 4 cores along with the corresponding average speedup against that of one core using intra-node parallelism. The streaming quality is improved due to the reduced encoding time (and hence the reduced access time). We achieve an average speedup of 1.8 times for 2 cores and 3.1 times for 4 cores for one GOP on a compute node. The deviation of the actual scaling from the ideal one (which is in linear proportion to M as defined in Table I) is caused by the sequential encoding part in our scheme. According to Amdahl's Law, we inferred that each GOP has a sequential processing part of around 2 sec.

To evaluate the performance of inter-node parallelism, we set p_e (defined in Section II.A) to be the maximum estimated GOP processing time. We compute corresponding d_i which will be used in both HM and LFM. We only take into account the lateness (i.e., the positive transcoding jitter) because it directly affects video freezes. In our evaluation, each compute node is configured with 4 cores for intra-node parallelism.

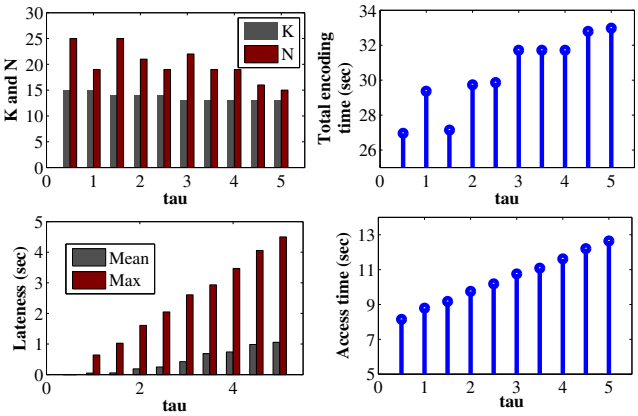

Fig. 6. HallSh-based mapping (HM) for different lateness upper bound $\tau = 0.5, 1, \ldots, 4.5, 5$.

Fig. 5 presents the LFM results using different number of compute nodes N (from 1 to 64). Due to the video content (and hence encoding time) heterogeneity, the actual encoding speedup deviates from the ideal case (which is in linear proportion to N). We set $R = 24$ fps and SG $= 8$, and along with GOP encoding time estimation, we compute from Eq. 5 that the minimum N is around 17-18. Fig. 5 presents the resulting maximum and mean lateness using LFM as well as the required minimal access time to avoid video freezes. It proves that the reduced computation power (smaller N) can degrade the GOP encoding lateness, and hence demands longer initial access time to accommodate transcoding jitters. Fig. 6 presents the HM results using different lateness upper bound τ from 0.5 to 5 sec. We show that HM can successfully decide the appropriate compute node number and limit the transcoding jitters. However the algorithm is not aimed at minimizing N. Hence its utilization of computation resources can be lower than LFM, which means that HM may require greater N in order to achieve the same level of lateness constraints (as illustrated in Fig. 5 and Fig. 6).

References

[1] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Transaction on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1103–1120, Sep. 2007.

[2] S. Winkler, "Video quality and beyond," in *Proceedings of European Signal Processing Conference*, Sep. 2007, pp. 150–153.

[3] ITU-P.910, "Subjective video quality assessment methods for multimedia applications," 2008.

[4] Y.-L. Huang, Y.-C. Shen, and J.-L. Wu, "Scalable computation for spatially scalable video coding using NVIDIA CUDA and multi-core CPU," in *Proc. of ACM Int'l Conference on Multimedia*, Oct. 2009, pp. 361–370.

[5] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," in *Proc. of ACM Symposium on Operating Systems Principles*, Oct. 2003, pp. 149–163.

[6] S. Sadjadi and et al, "A modeling approach for estimating execution time of long-running scientific applications," in *Proc. of IEEE Int'l Symposium on Parallel and Distributed Processing*, Apr. 2008, pp. 1–8.

[7] J. Chuzhoy, S. Guha, S. Khanna, and J. S. Naor, "Machine minimization for scheduling jobs with interval constraints," in *Proc. of IEEE Symposium on Foundations of Computer Science*, Oct. 2004, pp. 81–90.

[8] L. A. Hall and D. B. Shmoys, "Approximation schemes for constrained scheduling problems," in *Proc. of IEEE Symposium on Foundations of Computer Science*, Oct. 1989, pp. 134–13.