Optimizing the quality of scalable video streams on P2P Networks

Raj Kumar Rajendran, Dan Rubenstein
Dept. of Electrical Engineering
Columbia University, New York, NY 10025
Email: {raj,danr}@ee.columbia.edu

Abstract— **The volume of multimedia data, including video, served through Peer-to-Peer (P2P) networks is growing rapidly. Unfortunately, high bandwidth transfer rates are rarely available to P2P clients on a consistent basis. In addition, the rates are more variable and less predictable than in traditional client-server environments, making it difficult to use P2P networks to stream video for on-line viewing rather than for delayed playback.**

In this paper, we develop and evaluate on-line algorithms that coordinate the pre-fetching of scalably-coded variable bit-rate video. These algorithms are ideal for P2P environments in that they require no knowledge of the future variability or availability of bandwidth, yet produce a playback whose average rate and variability are comparable to the best off-line pre-fetching algorithms that have total future knowledge. To show this, we develop an off-line algorithm that provably optimizes quality and variability metrics. Using simulations based on actual P2P traces, we compare our on-line algorithms to the optimal off-line algorithm and find that our novel on-line algorithms exhibit near-optimal performance and significantly outperform more traditional pre-fetching methods.

I. Introduction

The volume of video-files served through Peer-to-Peer (P2P) networks is growing rapidly and users are often interested in streaming video for on-line viewing rather than for delayed playback. This requires a bandwidth rate ranging from 32 kbps to 300 kbps, which is rarely available consistently to P2P clients from a set of peers.

A practical solution that allows users with lower bandwidth availability to watch a video as it is downloaded involves the use of scalable-coding techniques. Using such techniques, the video can be encoded into a fixed number, M, of lower-rate streams called *layers* that are recombined to obtain a high fidelity copy of the video. Only the first layer is needed to decode and playback the video, but results in the poorest quality. As more layers are added, the quality improves until all M layers combine to produce the original video at full quality.

Since P2P systems require users to download content from other users who may leave during the viewing, the available download rate at a client can fluctuate greatly over time. Due to this, if the client were to always download as many layers as the current bandwidth rate allows for the current portion of the video, the quality of the playback would also fluctuate rapidly over time. Experimental studies [20] show that such fluctuations are more annoying than simply watching a lower-quality video at a more steady rate. Taking this into account, if all the bandwidth is used to pre-fetch a single layer at a time, the client would be forced to watch initial portions of the video at the lowest quality despite there being sufficient bandwidth to watch the entire video at a higher quality.

In this paper, we develop and evaluate on-line algorithms that coordinate the pre-fetching of scalably-coded variable bit-rate video components. Our algorithms are specifically designed for environments where the download rate varies unpredictably with time. These algorithms can be applied to current P2P systems which use multiple TCP streams to download content. In such systems the rate of download is a function of number of peers willing to serve the video at that time, the networking conditions and the manner in which TCP reacts to these conditions. The control of the downloading rate is outside the scope of the control of the application.

Our algorithms pre-fetch layers of future portions of the video in small chunks, as earlier portions are being played back. They do this with the aim of reducing the following metrics:

- **Waste**: the amount of bandwidth that was available but was not used to download portions of the video.
- **Smoothness**: the rate at which the quality of the playback (i.e., the number of scalably-coded layers used over a period of time) varies.
- **Variability**: the sum of the squares of the number of layers that are not used in the playback. This measure decreases as the variance in the number of layers used decreases, and also decreases when more layers appear in the playback.

We first design an **off-line** algorithm which, with knowledge of the future rate of the bandwidth channel, determines the pre-fetching strategy that minimizes the Waste and Variability metrics, and achieves near-minimal smoothness. We then construct three "Hill-building" **on-line** algorithms and compare their performance to both the optimal off-line algorithm and to more traditional on-line buffering algorithms. Our comparison uses *both simulated and real* bandwidth data. We collected actual traces of P2P downloads using a modified version of the Limewire Open source code. We find that our on-line algorithms are near-optimal in performance as judged by the metrics stated above, while more traditional methods perform significantly worse.

As far as we are aware, this problem of ordering the download of segments of scalably-coded coded videos in P2P networks to maximize the viewer's real-time viewing experience has not been addressed before. Related work by Ross, Saparilla and Cuestos [2], [13], [14] studies strategies for dividing bandwidth between the base and enhancement layers of a 2-layer video. They conclude that heuristics which take into account the amount of data remaining in the pre-fetch buffer outperform static divisions of bandwidth and videos with fewer changes in quality but slightly lower overall quality make for better viewing. Our work differs in that we consider a more general problem with a variable bitrate stream, N layers and employ quality measures that use first *and* second order variations. We also establish a theoretical performance bound that acts as a baseline for the comparison of the efficiencies of algorithms.

Work by Kim and Ammar [5] considers a layered video with finite buffers for each layer. Based on the size of each buffer, they determine the decision intervals for each layer that maximizes the smoothness of the layer. They take a conservative approach to the question of allocating bandwidth among layers and allocate all available bandwidth to the first layer, then to the second layer, and so on. Our work differs in that we assume, as is the case in today's P2P systems, that buffer space is relatively inexpensive and does not constrain the buffering strategy. We also optimize utilization and smoothness for all layers at once, rather than one layer at a time.

Several works also address the challenges of streaming a popular video simultaneously to numerous clients from a small number of broadcast servers [19], [7], [16], [1]. These works find methods that allow clients to view a few video streams at different but overlapping times.Our work differs in that we consider only a single receiver who must prefetch in order to cope with a download channel whose rate is unpredictable.

CoopNet [8] augments traditional client-server streaming with P2P streaming when the server is unable to handle increased demands due to flash crowds. CoopNet does not address the issue of variance in video quality nor does it use pre-fetching. Our results could easily be incorporated into a CoopNet system to improve the viewing experience.

[11] proposes layered coding with buffering as a solution to the problem of varying network bandwidth when streaming in a client-server environment. In contrast to our approach, they assume that that the congestion control mechanism is aware of the amount of video buffered and can be controlled by the application and therefore differ significantly in their solution.

The rest of the paper is organized as follows. Section II introduces Peer-to-Peer overlay networks and Scalable Coding while Section III quantifies our performance metrics and models the problem and solution. Section IV outlines the optimal off-line algorithm while Sections V and VI present our on-line scheduling algorithms and compare their performance to the off-line optimal algorithm and naive schedulers through simulations. Section VII outlines areas for further work and Section VIII concludes the paper.

II. Scalably Coded videos in Peer-to-Peer Networks

We use the existing P2P infrastructure as a guide for our network model. These P2P systems are distributed overlay networks without centralized control or hierarchical organization, where the software running at each node is equivalent. These networks are a good way to aggregate and use the large storage and bandwidth resources of idle individual computers. The decentralized, distributed nature of P2P systems make them robust against certain kinds of failures when used with carefully designed applications that take into account the continuous joining and leaving of nodes in the network. Our work focuses on the download component of the P2P network instead of the search component [15], [10], [18], [12], [3]. Unlike traditional client/server systems, the servers in P2P systems frequently start and stop participating in transmission, causing abrupt changes in the download rate at the receiver. In addition the TCP transport protocol is used, whose congestion control mechanism constantly varies the transfer rate of the connection with time.

Fine-Grained Scalable Coding (FGS) is widely available in current video codecs, and is now part of the MPEG-4 Standard. It is therefore being increasingly used in encoding the videos that exist on P2P networks. Such a scalable encoding consists of two layers: a small base-layer that is required to be transmitted and a much larger enhancement-layer that can optionally be transmitted as bandwidth becomes available. FGS allows the user to adjust the relative sizes of the base-layer and enhancement-layer and further allows the enhancement-layer to be broken up into an arbitrary number of hierarchical layers.

For our purposes, using the above fine-granular scalable-coding techniques, the video can be encoded into M identically-sized hierarchical layers by adjusting the relative sizes of the base and enhancement layers, and breaking up the enhancement layer into $M-1$ *identically-sized* layers. In such a layered encoding the first layer can be decoded by itself to play back the video at the lowest quality. Decoding an additional layer produces a video of better quality, and so on until decoding all M layers produces the best quality. The reader is referred to [4] and [9] for more details on MPEG-4 and FGS coding.

III. Problem Formulation

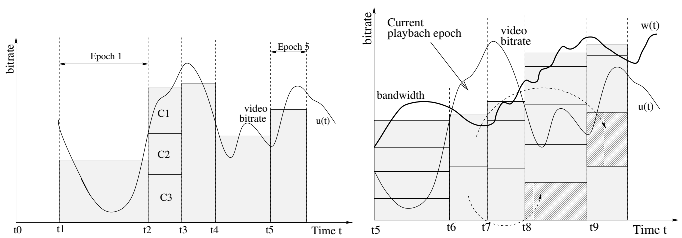
In this section we formulate the layer pre-fetching problem for which we will design solution algorithms. We begin by describing how the video segments, or *chunks*, the atoms of the video that will be pre-fetched, are constructed. We then formally define the metrics of interest in the context of these chunks, such that our optimization problems to minimize these metrics are well-posed.

A. The Model

Our model is discrete. We first partition the video along its running time axis into T *meta-chunks*, which are data blocks of a fixed and pre-determined byte size S (the Tth chunk may of course be smaller) [1]. These meta-chunks are numbered 1 through T in the order in which they are played out to view the video. We let t_i be the time into the video at which the ith meta-chunk must start playing. Hence, $t_1 = 0$, and if a user starts playback of the video at real clock time s_0 and does not pause, rewind or fast-forward, the ith meta-chunk begins playback at time $s_0 + t_i$. We refer to the time during which the ith meta-chunk is played back as the ith *epoch*. Note that since the video has a variable bitrate and the meta-chunks are all the same size in bytes, epoch times $(t_{i+1} - t_i)$ can vary with i. We also include a 0th pre-fetching epoch, where $t_i \leq 0$: the 0th epoch starts when the client begins downloading the video and ends at time $t_1 = 0$ when the client initiates playback of the video.

[1]We assume that epoch-lengths are longer than a GOP, and therefore different meta-chunks contribute equally to the quality of the video

As the video is divided into M equal-sized layers, each meta-chunk can be partitioned into M *chunks* of size S/M, where each chunk holds the video data of a given layer within that meta-chunk. These chunks form the atomic units of analysis in our model. In this formulation, the chunks associated with the ith meta-chunk of a video are played back in the ith epoch.



(A)Formation of chunks (B)Pre-fetching of chunks

Fig. 1. The Model

Figure 1(A) depicts a chunking of the video. Meta-chunks are the gray areas and are shown for the first through fifth epochs, and are placed within the epoch for which they are played back. While the rectangles are shaped differently in the different epochs, their areas are identical. The chunks of the meta-chunk within the second epoch are shown for a 3-layer scalably-coded encoding.

A chunk of video that is played back in epoch i must be downloaded in some earlier epoch $j, 0 \leq j < i \leq T$. In addition, as chunks are being played back in epoch i, chunks for a future epoch are being downloaded from sources using the network bandwidth that is available in epoch i. We refer to the bandwidth used during the ith epoch to prefetch chunks to be played back during future epochs as the *bandwidth slots of epoch i* where each slot can be used to download exactly one chunk. Since epochs can last for different lengths of time, and since the bandwidth streaming rate from the sources to the receiver varies with time, the number of bandwidth slots within an epoch varies. Note that it is unlikely that the bandwidth available during an epoch is an exact integer multiple of the chunk size. This extra bandwidth can be "rolled over" to the subsequent epoch. More formally, if W_i is the total number of bytes that can be downloaded by the end of the ith epoch, then the number of slots for epoch i is $\lfloor W_i/S \rfloor - \lfloor W_{i-1}/S \rfloor$, with $W_{-1} = 0$.

Note that our model is easily extended to account for pauses and rewinds of the video: the start time of an epoch i is simply the actual clock time that elapses between when video playback first starts and when the chunks for epoch i are first needed. Since pausing and rewinding can only extend this time, doing so can only increase the number of bandwidth slots that transpire by the time the ith epoch is to be played back.

Figure 1(B) depicts the number of chunks that could be downloaded per epoch. In epochs 5,6,7,8, and 9, the available bandwidth permitted the download of 4,2,2,4 and 3 chunks respectively, [2] The current playback point of the video is within the sixth epoch. The arrows pointing from the chunks in the sixth epoch toward the eighth and ninth epoch are meant to indicate that the bandwidth available during the sixth epoch is used to pre-fetch two chunks from the eighth and ninth epochs. The download-rate $w(t)$ and playback-rate $u(t)$ are also illustrated in Figure 1(B). The number of chunks that could be downloaded is proportional to the area under the curve $w(t)$ for that epoch. Details about the practicalities of how the chunks are formed and how their download is coordinated is describe in detail in [17].

We define an *available bandwidth vector*, $W = \langle w_0, w_1, \cdots, \ w_{T-1} \rangle$, where w_i is the number of bandwidth slots available during the ith epoch, $0 \leq i < T$. An *allocation* is also a T-component vector, $A = \langle a_1, \ldots, a_T \rangle$, that indicates the number of chunks that are played out during the ith epoch, $0 < i \leq T$. An allocation $A = \langle a_1, \ldots, a_T \rangle$ is called *feasible under available bandwidth vector W* if there exists a pre-fetching strategy under the bandwidth constraints described by W that permits a_i chunks to be used in the playback of the ith epoch for all $1 \leq i \leq T$. Feasibility ensures that we only consider allocations that can be constructed where bandwidth slots from epoch i are used to pre-fetch data for future epochs $j, j > i$.

A pre-fetching algorithm takes as input an available bandwidth vector W and outputs an allocation A. An off-line pre-fetching algorithm may view the entire input at once. An on-line algorithm iterates epoch by epoch, and only knows the value of the ith component of the vector after the ith epoch has completed. During the ith epoch, the only chunks it can download that can be used for playback are those that lie in epochs $j > i$, hence, once it sees the ith component w_i of W, it has entered the $i + 1$st epoch and can therefore only modify components $a_j, j > i + 1$ in the allocation A.

Note that, from the perspective of our model, since partially downloaded chunks may not be used in the playback of the video, it never makes sense for an on-line algorithm to simultaneously download multiple chunks. Instead of downloading r chunks in parallel, all bandwidth resources would be better served focusing the download bandwidth on a single chunk and downloading these r chunks in sequence (we discuss the practical details of doing this in [17]). In addition, since future bandwidth rates are unknown, there is no reason for an on-line algorithm to "plan ahead" beyond the current chunk it is downloading: there is no loss in performance by deciding the next chunk to download only at the time when that download must commence. Note also that the download of a chunk to be played back in epoch i (which presumably was started while playback was in some epoch $j < i$) is terminated immediately if the download is not complete when the playback enters epoch i, since then it cannot be used within the playout.

B. Performance Measures

The performance measures we use in this paper are motivated by a perceptual-quality study that validates our intuitive notions of video quality [20]. The work concludes that the important indicator of video quality is the average quality of each image as measured by the number of bits used. However, it is important to consider second-order changes: given two encodings with the same average quality, the study shows that users clearly prefer the encoding with smaller changes

[2]The figure also illustrates partial chunks in 7,8 and 9, which for simplicity of analysis are not used.

in quality. Such variations can be captured by measures such as variance and the average change in pictorial-quality. The subjective study further shows that users are more sensitive to quick decreases in quality than increases. We use this observation in selecting performance measures, and later in the design of our on-line algorithms.

Of the many possible measures that capture the above mentioned qualities of a video, we limit ourselves to three that we believe capture the essential first and second-order variations. These are a first-order measure we call waste that quantifies the utilization of available bandwidth, a first plus second order measure called variability that measures both utilization and variance in quality, and a second-order measure called smoothness that captures changes in the quality between consecutive epochs of the video. It must be noted that these measures relate to perceptual quality in not well understood non-linear fashions, and therefore cannot be compared to one another. However they clearly indicate the efficiency of one algorithm vis-a-vis one another.

We now define our measures, all of which take as input an allocation A = $\langle a_i, \ldots, a_T \rangle$ under an available bandwidth vector W.

$$\textbf{Waste } w(A) = \max_{B \in F(W)} (\sum_{j=1}^{T} b_j) - \sum_{i=1}^{T} a_i$$

where $F(W)$ is the set of all feasible allocations under W. That is, we measure an allocation's waste by comparing it to the allocation with the best possible utilization. An allocator wastes bandwidth when it has been too conservative in its allocation of early chunks and it finds that all M chunks for all future epochs have already been pre-fetched despite having available bandwidth.

$$\textbf{Variability } \mathcal{V}(A) = \sum_{i=1}^{T} (M - a_i)^2$$

Intuitively, a video stream that has near constant quality will be more pleasant to view than one with large swings in quality. What we wish is a measure that increases when the variance of the number of layers used goes up, and decreases when more layers appear in the playback so that minimizing the metric reduces the variance and improves the mean. Variability satisfies this requirement.

$$\textbf{Smoothness } s(A) = \sum_{i=2}^{T} abs(a_{i-1} - a_i)$$

Consider two video streams of six seconds each where the number of layers viewed in each of the six seconds is $\langle 1, 2, 1, 2, 1, 2 \rangle$ for the first video and $\langle 1, 1, 1, 2, 2, 2 \rangle$ for the second. The Variability values of their qualities is the same but we will clearly prefer the second video over the first, as it will have fewer changes in quality. Smoothness captures this preference.

IV. An Optimal Off-line Algorithm

In this section we develop an off-line algorithm, Best-Allocator, that allocates chunks to epochs, creating an ordered allocation $A = \langle a_1, a_2, \cdots, a_T \rangle$. This allocation minimizes waste and variability metrics and has near-minimum smoothness (i.e., no larger than M, the number of layers) in comparison to all other feasible allocations. While this optimal off-line algorithm cannot be used in practice, it can be used to gauge the performance of the on-line algorithms we introduce in the next section.

Best-Allocator is fed as its input an available bandwidth vector $W = \langle w_0, w_1, \cdots, w_{T-1} \rangle$. Recall that a chunk that is to be played back in epoch i must be downloaded during an epoch j where $j < i$. Hence each bandwidth slot of epoch i should only be used to download a chunk from epoch $j > i$, when there exist such chunks that have not already been assigned to earlier bandwidth slots. The algorithm proceeds over multiple iterations. In each iteration, a single bandwidth slot is considered, and it is either assigned a chunk, or else is not used to pre-fetch data for live viewing. The algorithm proceeds over the bandwidth slots starting with those in the $T - 1$st epoch, and works its way backward to the 0th epoch. The chunk to which that slot is assigned is drawn from a subsequent epoch with the fewest number of chunks to which bandwidth slots have already been assigned. If two or more subsequent epochs ties for the minimum, the ties are broken by choosing the later epoch. If all subsequent epochs' chunks are already allocated, then the current bandwidth slot under consideration is not used for pre-fetching.

More formally, let the vector $A(j) = \langle a_1(j), a_2(j), \cdots, a_T(j) \rangle$ represent the number of chunks in each epoch that have been assigned a download slot after the jth iteration of the algorithm. Let $X(j) = \langle x_0(j), x_1(j), \cdots, x_{T-1}(j) \rangle$ be the numbers of bandwidth slots that have not yet been assigned in epochs 0 through $T - 1$. Note that $A(0) = \langle 0, 0, \cdots, 0 \rangle$ and $X(0) = W$.

During the jth iteration, we perform the following, stopping only when $X(j) = \langle 0, 0, \cdots, 0 \rangle$:

1) If all $x(j) > 0$ set $k = T$. Else set $k = \ell$ that satisfies $x_\ell(j) > 0$ and $x_m(j) = 0$ for all $m > \ell$.
2) Set $X(j+1) = \langle x_0(j), x_1(j), \cdots, x_{k-1}(j), x_k(j) - 1, 0, 0, \cdots, 0 \rangle$
3) Set n equal to the largest ℓ that satisfies both $\ell > k$ and $a_\ell(j) = \min_{m>k} a_m(j)$
4) If $a_n(j) = M$ do nothing. Else $A(j+1) = \langle a_0(j), a_1(j), \cdots, a_{n-1}(j), a_n(j) + 1, a_{n+1}(j), \cdots, a_T(j) \rangle$.

We now state the most important results about the optimality of the allocation produced by Best-Allocator without proof. For proofs and other results please refer to [17].

Claim 4.1: The allocation formed by Best-Allocator minimizes waste.

Claim 4.2: The smoothness of allocation A formed by Best-Allocator is $a_T - a_0$ and no allocation that uses a_T layers in any epoch can have lower smoothness.

Claim 4.3: Let A be the allocation generated by Best-Allocator and let B be any other feasible allocation with identical waste. Then $Variability(A) \leq Variability(B)$.

V. On-line allocation algorithms

In this section we present five on-line bin-packing algorithms that can be used to schedule the downloads of scalably-coded videos. The off-line algorithm "Best-Allocator" presented in Sec IV achieves the best possible performance for the given input but makes its allocation decisions *after* receiving all inputs. In reality we will not know future bandwidth rates, therefore on-line algorithms must be designed to make decision without knowing future bandwidth.

Given that the bandwidth at the client varies, the scheduler at the client is faced with a choice in each epoch t: whether to use its bandwidth to download and display as many chunks of the epochs immediately following the current active epoch, thereby greedily maximizing current pictorial-quality, or to use the current bandwidth to download as much of a single layer for current and future epochs, so that if bandwidth in future epochs prove insufficient, these chunks of data will provide at least the minimum quality and thereby minimize variance in the quality of the video.

A. Naive allocators

First consider the two allocators that exemplify the two extreme ends of this tradeoff.

- **[Same-Index]** This allocator allocates all bandwidth to downloading chunks belonging to the current or nearest future epoch for which it has not yet downloaded all the chunks. This allocator will tend not to waste capacity as it greedily uses it up.
- **[Smallest-Bin]** This allocator allocates all bandwidth slots to the epoch with fewest layers already downloaded, breaking ties by choosing the earliest epoch. Such an allocation strategy will first download *all of layer-1* of the video, then *all of layer-2* and so on. This approach will produce unchanging and smooth quality, but will waste capacity.

B. Constrained allocators

Intuitively we would like algorithms that operate somewhere between the two extremes of the same-index and smallest-bin allocators. Our solution are algorithms that maximize current quality, but with smoothness constraints derived from perceptual quality studies [20] that indicate that users dislike quick changes in video quality, particularly *decreases* in quality. Our algorithms do so by constraining the **downhill slope** of an allocation. Within this restriction the different on-line algorithms in our suite attempt to maximize different desirable qualities. More formally, constrained downhill slope allocators build allocation B = $\langle b_1, \ldots, b_T \rangle$ such that at any point in the building process, $b_i - b_{i+1} < C, 0 \leq i < T$ for some integer constraint C.

We present three such allocators. In describing the algorithms we will assume that the current epoch is t_i and the algorithm is allocating a chunk of bandwidth from the bandwidth slot in epoch t_i to a chunk in some epoch $t_j, j > i$, where b_i indicates the number of chunks allocated to epoch t_i. To help the reader visualize this process, consider having an empty "bin" B_j assigned to hold the chunks in t_j that have been allocated for download using a slot from a previous epoch, t_i. b_j is the number of chunks in bin B_j, and each time a bandwidth slot (from a previous epoch) is allocated to serve a chunk from epoch t_j, b_j is incremented.

- **[Largest-Hill]** This algorithm allocates each chunk of bandwidth to the bin B_j with the smallest index j such that the constraint $b_j - b_{j+1} < C$ is satisfied after the chunk has been allocated. The largest-hill allocator attempts to maximize the size of the earliest bin possible while maintaining the slope constraint. Such a strategy tends to produce "hills" of allocations with a constant downhill slope; thereby the name.
- **[Mean-Hill]** Given that the average bandwidth seen so far is μ_w chunks per epoch, the Mean-Hill allocator uses the following rules to allocate each chunk of bandwidth.
  - Find the bin B_j with the smallest index j such that the slope constraint $(b_j - b_{j+1}) < C$ is satisfied.
  - If the size of this bin b_j is less than μ_w, allocate the chunk to this bin B_j.
  - Else, allocate the chunk to the most empty bin B_m. In case of ties allocate the chunk to the most empty bin with the smallest index (the earliest-bin).

  The Mean-Hill allocator attempts to maximize the current bin-size while ensuring that the slope-constraint is satisfied. Once the current bin has grown bigger than μ_w, it uses its bandwidth to download the full video one layer at a time. This algorithm operates under the assumption of mean-reversion; that larger-than-mean current bandwidth will be compensated by smaller-than-mean bandwidth in the future.
- **[Widest-Hill]** This allocator allocates each chunk to the bin with the smallest index j that satisfies the slope constraint $(b_j - b_{j+1}) < C$ **and** the height constraint $b_j \leq \mu_w$. This strategy tends to produce allocations that first grow up to the mean, then widen while satisfying the slope constraint.

Fig. 2 shows typical allocations of the three algorithms with the slope-constraint $C = 1$, input $\langle 6, 7, 9, 11 \rangle$ and $\mu_w = 3$. The Largest-Hill algorithm grows tall hills while maintaining the downhill slope. The Wide-hill grows hills until they reach the mean, then widens them while maintaining the slope constraint, and the Mean-Hill algorithm grow hills up to the mean, then uses excess bandwidth to fill out future bins.

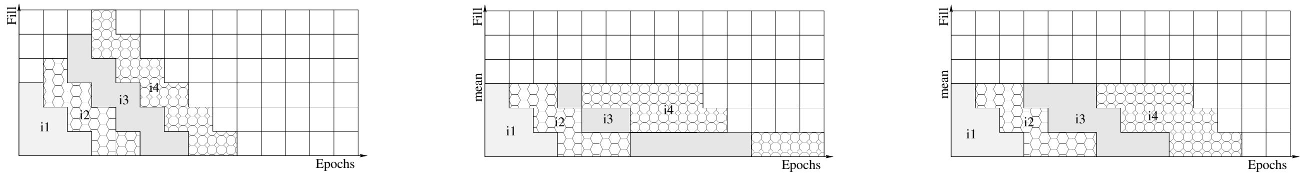
VI. Results

In this section we present and evaluate the performance of the off-line Best-Allocator algorithm and five on-line bin-packing algorithms. Comparisons are achieved by means of two simulations which look at the ability of the five on-line algorithms and the Best-Allocator to minimize Variability [3], smoothness and waste.

In the first simulation experiment we chart these measures as functions of the mean and the variance of the input. Such a study will show us how the algorithms perform under two interesting conditions: when the mean of the bandwidth approaches the mean bit-rate of the video, and secondly as the network bandwidth shows increasing fluctuations.

In the second trace experiment we use bandwidth traces obtained by the authors while downloading videos from the Gnutella network as input. In this simulation we study the performance as a function of the average epoch length. This study will indicate the performance of the algorithms in real-life bandwidth conditions, and indicate appropriate decision-intervals.

[3] We chart the square-root of the Variability rather than the Variability itself, as the square-root has units of $chunks$, and is easier to visualize than Variability, which has units of $chunks^2$.



(A) Largest-Hill (B) Mean-Hill (C) Wide-Hill

Fig. 2. Three slope-constrained algorithms

For our experiments we assume that $u(t)$ the video-bitrate is a known constant function. In the simulation experiment $w(t)$ is generated for each epoch by drawing randomly from the distribution specified. The experiments were conducted for 600 epochs, and the result averaged over 100 runs. In the trace experiment $w(t)$ was derived each second by counting the number of bytes of a video that was downloaded in that second. We use two traces which lasted approximately 3,600 and 11,000 seconds. The chunk-size was chosen such that the overall mean of the bandwidth corresponded to $M/2$ chunks and the bandwidths in each epoch were rounded to an integer number of chunks. Throughout the experiments a value of 6 was used for M.

A. Simulated bandwidth

We first studied the performance of the various algorithms as the available bandwidth approaches the maximum video bandwidth. To do this we independently generated the input bandwidth from a uniform distribution and gradually increased the mean of the uniform distribution (note that this increases the standard-deviation as well) to the maximum video bandwidth. The performances of the algorithms as measured by the three metrics are plotted as a function of the mean in Fig. 3.

It can be seen from Fig. 3 that constrained-slope allocators vastly outperform the naive algorithms. It can be seen that all three constrained-slope algorithms have robust performances when the input bandwidth is half the maximum video-bandwidth while the Mean-Hill algorithm is marginally more robust than the Largest-Hill and Wide-Hill algorithm as the available input bandwidth approaches the maximum video bandwidth.

We also studied the performance of the different algorithms as the variance of the input bandwidth increases while the mean remains the same by using a Gaussian distribution. Such an analysis will allow us to choose the optimal algorithm for different bandwidth environments. The results of this experiment are presented in [17].

In summary the novel on-line algorithms provide very good performance compared to the ideal offline case, and vastly outperform naive strategies.

B. Bandwidth Traces

In this section we present the results of applying the algorithms to real data. Two sets of traces were obtained, the first through a DSL line and the second on a T1 network. We created a program, based on modifying the Limewire Open source code [6], that continually downloaded videos from the Gnutella network, and traced the aggregate bandwidths to the servers each second. From these two sets of data we picked one representative trace each: the first, from the T1 network lasted 3,663 seconds, and the second from the DSL connection lasted 11,682 seconds. We then computed the performance of the on-line algorithms when applied to these traces for an epoch-lengths of 1, 2,4,8,16,32 and 128 seconds, to study the effect of time-scales on the performance of the heuristic algorithms. We noticed that the traces showed large bandwidth variation across all scales.

The first trace was run on a computer connected to the Internet through a telephone line using DSL. The trace lasted for 11,682 seconds and downloaded a 80 MB video. Two servers were serving the video simultaneously for large parts of the download. The algorithms were run on this trace for different average epoch lengths of 1, 2, 4, 8, 16, 32, 64 and 128 seconds and the standard deviation of the bandwidth reduced from 2.97 to 1.64 in response. The performances are charted in Fig. 4 as a function of these different standard-deviations resulting from varying the average epoch-lengths.

A second trace was run on a computer connected to the Internet through a T1 connection. Details of this experiment can be seen in [17].

With both the DSL connection and the T1 connection, the constrained-slope allocators vastly outperformed the naive allocators. The Mean-Hill and Wide-Hill performed close to the bound provided by the best-allocator. As the variance of the input bandwidth increased (corresponding to smaller epoch-lengths) the Mean-Hill and the Wide-Hill algorithms showed consistent performance, while the Largest-Hill algorithm slightly lagged in performance. Overall, the Mean-Hill and Wide-Hill algorithms provide near-optimal and consistent performance in real-life bandwidth scenarios across varying epoch-lengths.

VII. Issues

Our approach can easily be extended to handle rewinds while the Mean-Hill algorithm can be adapted to support fast-forwarding. We plan to work next on scheduling under unequal sized layers, and determining the optimal pre-fetch interval before the start of playback. We also plan to implement our algorithms as an application and verify that they do indeed produce the most even quality relative to other video-streaming algorithms.

VIII. Conclusion

Peer-to-Peer networks are increasingly being used to stream videos where often the user wishes to view a video at a bandwidth larger than that obtainable in current P2P systems. Scalably Coded video is an attractive solution to this problem. We show that in practical P2P settings, because the user's

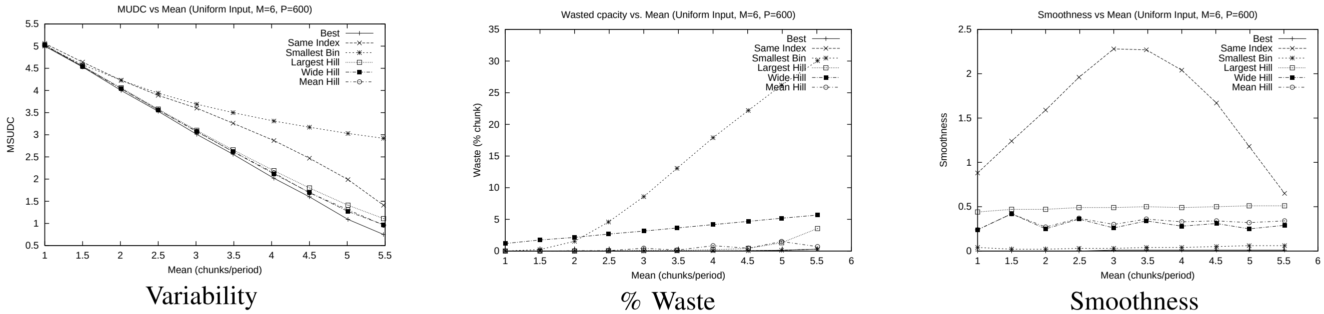


Fig. 3. Uniform Input

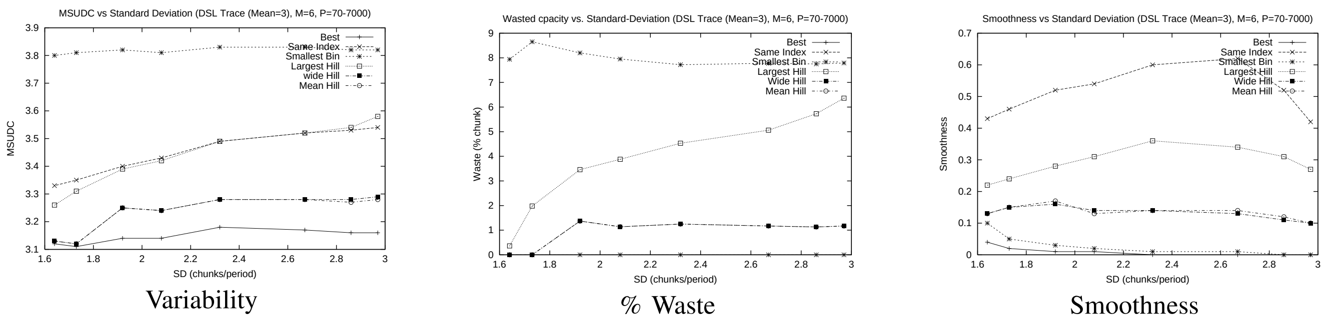


Fig. 4. DSL Input

bandwidth to multiple servers will vary widely, it is imperative to pre-fetch downloads to ensure uninterrupted smooth viewing and that the quality of the video is sharply affected by the algorithm used. We present bounds on the performance that can be achieved by developing an optimal offline algorithm, then present on-line algorithms that vastly outperform naive schedulers. Through simulations we show that our solutions perform close to the best possible performance.

ACKNOWLEDGMENTS

This material was supported in part by the National Science Foundation under Grants No. CAREER ANI-0133829, and ITR ANI-0325495, by a gift from the Intel Information Technology Research Council, and by an IBM Faculty Partnership Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] E. Coffman, P. Jelenkovic, and P. Momcilovic. The Dyadic Stream Merging Algorithm. *Journal of Algorithms*, 43(1), 2002.
[2] P. de Cuestos and K. W. Ross. Adaptive rate control for streaming stored fine-grained scalable video. *NOSSDAV*, 2002.
[3] The Gnutella Protocol Specification v0.4, revision 1.2. Available from http://gnutella.wego.com.
[4] Overview of the mpeg-4 standard, 2002. Available at http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm.
[5] T. Kim and M. Ammar. Quality adaptation for mpeg-4 fine grained scalable video. In *INFOCOM*, 2003.
[6] Limewire open-source software. Available from http://www.limewire.org/project/www/Docs.html.
[7] A. Mahanti, D. Eager, M. Vernon, and D. Sundaram-Stukel. Scalable On-Demand Media Streaming with Packet Loss Recovery. *Transactions on Networking*, 11(2), April 2003.
[8] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. *ACM/IEEE NOSSDAV*, May 2002.
[9] H. Radha, M. van der Schaar, and Y. Chen. The mpeg-4 fine-grained scalable video coding method for multimedia streaming over ip. *IEEE Trans. on Multimedia*, March 2001.
[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, August 2001.
[11] R. Rejaie, M. Handley, and D. Estrin. Quality adaptation for congestion controlled video playback over the internet. In *SIGCOMM*, pages 189–200, 1999.
[12] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of ACM SOSP'01*, Banff, Canada, October 2001.
[13] D. Saparilla and K. W. Ross. Optimal streaming of layered encoded video. *INFOCOM*, 2000.
[14] D. Saparilla and K. W. Ross. Streaming stored continuous media over fair-share bandwidth. *NOSSDAV*, 2002.
[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, August 2001.
[16] H. Tan, D. Eager, and M. Vernon. Delimiting the Effectiveness of Scalable Streaming Protocols. In *Proc. IFIP WG 7.3 22nd Int'l. Symp. on Computer Performance Modeling and Evaluation (Performance)*, Rome, Italy, September 2002.
[17] Optimizing the quality of scalable video streams on p2p networks. Available from http://www.ee.columbia.edu/ kumar/papers/p2004-01.pdf.
[18] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, UC Berkeley, April 2001.
[19] Y. Zhao, D. Eager, and M. Vernon. Network Bandwidth Requirements for Scalable On-Demand Streaming. In *Proceedings of IEEE INFOCOM'02*, New York, NY, June 2002.
[20] M. Zink, O. Kunzel, J. Schmitt, and R. Steinmetz. Subjective impression of variations in layer encoded videos. *International Workshop on Quality of Service*, pages 137–154, 2003.