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.

Key words: P2P, Video, Streaming, Quality, Scheduling

1 Introduction

The volume of multimedia data, including video, served through Peer-to-Peer (P2P) networks is growing rapidly. Users are often interested in streaming video for on-line viewing rather than for delayed playback, which requires a bandwidth rate ranging from 32 kbps to 300 kbps. Unfortunately, such high bandwidth transfer rates are rarely available to P2P clients from a set of peers on a consistent basis.

A practical solution that allows users with lower bandwidth receiving rates to watch the 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 can be combined to produce the original video at full quality. Modern scalable-coding techniques such as Fine-Grained Scalable coding (FGS) allow the video to be broken up so that each of the M layers requires a bandwidth u_M that is a constant fraction of the bandwidth u of the original video.

Since P2P systems require users to download content from other users, the available download rate at a client fluctuates greatly over time. 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 fluctuate rapidly over time. Experimental studies [1] show that such fluctuations are considered more annoying than simply watching a lower-quality video at a more steady rate. On the other hand, if all the bandwidth is used to pre-fetch a single layer at a time, then the client would be forced to watch a large portion of the initial video at the lowest quality when often there is 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. Such algorithms are amenable within current P2P systems which use multiple TCP streams to download content. In these 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, with an aim of reducing the following metrics:

- **Waste**: the amount of bandwidth that was available but was not used to download portions of the video that were included in the playback.
- **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 is decreased, 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 of layers that minimizes the Waste and Variability metrics, and achieves near-minimal smoothness. We then construct three "Hill-building" on-line algorithms and compare their per-

formance to both the optimal off-line algorithm and to more traditional on-line buffering algorithms. Our comparison uses **both simulated and real** bandwidth data. Actual traces of P2P downloads were collected by us using a modified version of the Limewire Open source code. We find that, in the context of the above metrics, our on-line algorithms are near-optimal while the more traditional methods perform significantly worse than this optimal pre-fetching strategy.

1.1 Prior Work

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.

The work that most closely relates to our work [2,3] is by Ross, Saparilla and Cuestos [4–6] where the authors study strategies for dividing available bandwidth between the base-layer and enhancement-layer of a two-layered stream. They conclude that heuristics which take into account the amount of data remaining in the pre-fetch buffer outperform static divisions of bandwidth. They also conclude that maximizing just the overall amount of data streamed causes undesirable fluctuations in quality, and provide additional heuristics that produce a smoother stream while minimally reducing the overall quality. Our work complements and differs from their work in a number of ways: we consider a more general problem with a variable bitrate stream and N layers rather than a constant bitrate stream with two layers, and our quality measures take into account first *and* second order variations, borne out in subjective experiments detailed in [1].

In addition we abstract the problem into a discrete rather than continuous model, and most importantly, establish a theoretical performance bound that acts as a baseline for the comparison of the efficiencies of algorithms. Furthermore our work considers the problem in the context of P2P networks where bandwidth is much less predictable than traditional client-server environments and our on-line algorithms are designed specifically with such variations in mind. For instance, we refrain from making the assumption that the long-term average download rate is known in advance, and our simulations specifically use traces obtained from P2P networks.

A second work that is closely related is by Kim and Ammar [7] which 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 that buffer space is relatively inexpensive, and therefore infinite implying constant decision-intervals. Secondly we optimize utilization and smoothness for all layers at once, rather than

one layer at a time. Another related work is from Taal, Pouwelse and Lagendijk [8] , where the authors describe a system for multicasting a video over a P2P network using Multiple Description Coding. Their system aims to maximize the average quality experienced by all clients. Unlike them we concern ourselves with unicast streams.

Several works also address the challenges of streaming a popular video simultaneously to numerous clients from a small number of broadcast servers using techniques such as skimming [9], periodic broadcast [10], and stream merging [11,12]. These works assume that the server's transmission rate is predictable, and that the major challenge is to find methods that allow clients that wish to view the video at different but overlapping times to pre-fetch portions from the same transmission. Our work differs in that we consider only a single receiver who must pre-fetch in order to cope with a download channel whose rate is unpredictable.

CoopNet [13] augments traditional client-server streaming with P2P streaming when the server is unable to handle the increased demands due to flash crowds. In CoopNet clients cache parts of recently viewed streams, and stream them if the server is overwhelmed. They use Multiple-Description Coding to improve robustness in the wake of the frequent joining and leaving of peers. CoopNet does not address the issue of variance in video quality and does not do pre-fetching. We suspect that our results could easily be worked into a CoopNet system to improve the viewing experience.

[14] propose layered coding with buffering as a solution to the problem of varying network bandwidth while streaming in a client-server environment. Buffering is used to improve video quality, but it is assumed in this work that the congestion control mechanism is aware of the amount of video buffered and can be controlled. In contrast, our work does not attempt to control the bandwidth rate, but rather builds on **top** of existing P2P systems that already have mechanisms such as multiple TCP streams, that dictate the rate. We maximize quality purely by making decisions on the order in which segment of a video stream are downloaded, a subtle but important difference.

In addition practical multi-chunk P2P streaming systems like BitTorrent [15] and Coolstreaming [16] exist and are very popular. However these systems aim to optimize the download of *generic* files and videos through chunking and peer-downloads. We deal with the more specific problem of streaming layered video.

The rest of the paper is organized as follows. Section 2 briefly introduce Peer-to-Peer overlay networks and Scalable Coding. In Section 3 we quantify our performance metrics, and model the problem and its solutions. In Section 4 we present performance bounds on the solutions to the problem. Section 5 presents our on-line scheduling algorithms, then compares their performance to the bounds and of naive schedulers through the use of simulations in Section 6. In Section 7 we list some

issues for further consideration, and conclude the paper in Section 8.

2 Scalably Coded videos in Peer-to-Peer Networks

We use the existing P2P infrastructure as a guide for our network model. 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 continuously joining and leaving of nodes in the network.

Our work focuses on the download component of the P2P network. Our results are independent of the search component used to identify sources for download, so any existing searching method applies [17–21]. What is important is that, unlike traditional client/server paradigms, serving sources frequently start and stop participating in the transmission, causing abrupt changes in the download rate at the receiver. In addition, like the traditional paradigm, the individual connections utilize the TCP transport protocol whose congestion control mechanism constantly varies the transfer rate of the connection.

2.1 Scalable Coding

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 requires to be transmitted and a much larger enhancement-layer that can 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 [22] and [23] for more details on MPEG-4 and FGS coding.

3 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. We summarize the symbols and terms used in this section in Table 1 for easy reference.

3.1 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 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.

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.

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. The curve $u(t)$ is explained later in this section.

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

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

s	clock-time
t	video time
$\tau(t)$	the actual clock-time at which the viewer watches the tth second of the video
$b(s)$	rate at which bandwidth is available to the client at clock time s
$u(t)$	playback-rate or the rate at which playback must occur to view the tth second of the video at full quality
$w(t)$	download rate
M	number of layers in video
$chunk$	atomic unit of video used in our model = S/M bytes where S is chosen appropriately
$epoch$	time it takes to playback a chunk of video
T	total number of epochs in video
t_i	video time when ith epoch begins
bandwidth-slot	bandwidth to download 1 chunk of video data
w_i	number of chunks of bandwidth or *bandwidth-slots* that arrive during epoch i
$W = \langle w_0, \ldots, w_{T-1} \rangle$	Discretized bandwidth as a vector of bandwidth-slots
$A = \langle a_1, \ldots, a_T \rangle$	An *allocation*, where a_i is the number of chunks used in the playout of the ith epoch of the video

Table 1
Table of terms and symbols

bandwidth that is available in epoch i. We refer to the bandwidth used during the ith epoch to pre-fetch chunks to be played back during future epochs as the *bandwidth slots of epoch* i where each slot can be used to download at most 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.

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 is said to be *ordered* when $a_1 \leq a_2 \leq \ldots \leq a_T$. 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 $1 \leq i \leq T$. We can simply say that an allocation A is feasible when the available bandwidth vector can be determined from the context. 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 is given as input an available bandwidth vector W and outputs an allocation. An off-line pre-fetching algorithm may view the entire input at once. An on-line algorithm iterates epoch by epoch, and can 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

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

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 downloading these r chunks in sequence (the practical details of doing this are discussed in Section 3.1.2). In addition, 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.

3.1.1 Constructing the Epoch View

The above description of our model is sufficient for the reader to proceed to subsection 3.2. However, some readers may be concerned with the practicalities of how the chunks are formed and how their download is coordinated. We describe these details here:

We assume that the video is a variable bit-rate encoding, with $u(t)$ indicating the **playback-rate**: the rate at which the playback must occur to view the tth second of the video at full quality. Each layer is therefore played back at time t at rate $u(t)/M$. Each meta-chunk contains the same pre-determined number of bytes, S. The epoch times t_i are then determined recursively by solving $S = \int_{t=t_{i-1}}^{t_i} u(t)dt$, with $t_1 = 0$. Note that if the video is constant bit-rate ($u(t) = c$), the epoch lengths are the same. In Figure 1, the curvy line indicates the playback-rate of the video, $u(t)$. The area of the rectangle in the ith epoch matches the area under the $u(t)$ curve within that epoch.

Having formally discussed the terminology necessary to describe the playout time of the video layers, we now turn our focus toward the terminology needed to discuss the pre-fetching of the video layers. We assume that all sources that transmit chunks have complete encodings of the video. This is not an unreasonable assumption for a P2P network, since often a client does not become a source for an object until it has received the entire object. We use $W(t)$ to represent the **aggregate number of bytes** that the collection of sources have forwarded to the client at the time when the client is viewing the tth second of the video. More formally, if $\tau(t)$ indicates the actual clock-time at which the viewer watches the tth second of the video, $b(s)$ is the rate at which bandwidth is available to the client at actual clock time s, and the user initiates the download at actual clock time s_0, then $W(t) = \int_{s=s_0}^{\tau(t)} b(s)ds$. Note that it is possible for $W(0) = A > 0$, i.e., the client downloads A bytes of data prior to initiating the viewing of the video. The period of time in which this pre-fetching occurs is referred to as the zeroth epoch. Recall that the time t_0 has a value $x \leq 0$ when the user starts playback of the video x seconds after the download

is initiated. $W(t)$ is clearly non-decreasing with t (assuming the client does not rewind), but need not be piecewise continuous. For instance, pauses in viewing can cause discontinuities.

Note that, while $u(t)$ is known in advance, $w(t)$ is not, both because the user may pause the video and because the aggregate downloading rate from the transmission sources is unpredictable. Hence, the on-line scheduling problem involves determining the order in which to download chunks to maximize perceived quality, given a known $u(t)$ and an unknown $W(t)$.

The download-rate $w(t) = \dot{W}(t)$ 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.

3.1.2 Coordinating download of a single chunk

Last, let us discuss how a single chunk can be downloaded efficiently when several distributed sources are transmitting the video data to the peer receiver. First, the number of bytes, S/M allocated to each chunk should be fairly large, on the order of 100 KB, which represents roughly 5 seconds of video playback. This is sufficient time to contact all senders to inform them to focus on the download of a specific chunk. The chunk itself can then be partitioned into micro-chunks where each micro-chunk fits within a single packet, and the different micro-chunks can then be scheduled across the multiple servers and delivered using standard techniques [24], or in a more efficient manner using parity encoding techniques [25]. If a server departs, the former scheme must detect the departure and reassign the micro-chunks that were assigned to the departing server to another server. In the latter, no reassignment is needed. Any inefficiencies in downloading can easily be incorporated into our model simply reducing the available bandwidth $w(t)$. When the download of a chunk is complete, the receiver can indicate to the sources the next chunk to download.

In practice, to maximize efficiency and reduce overlap, it may be worthwhile to assign different senders to different chunks. However, one must deal with the possibility that a sudden drop in bandwidth availability could leave a receiver with multiple partially downloaded chunks. Evaluating this tradeoff and cost is beyond the scope of this paper.

3.2 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 [1]. The most important indicator is the average quality of each image, as measured by the aver-

age number of bits used in each image or the average signal-to-noise ratio (PSNR). However, it is important to consider second-order changes: given two encodings with the same average quality, the perceptual-quality study shows that users clearly prefer the encoding with smaller changes 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 while being less sensitive to increases. We use this information in the selection of our performance measures, and later in the design of our online algorithms.

Of the many possible measures that capture the above mentioned effects, we limit ourselves to three that we believe capture the essential first and second-order variations. They 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. While using these measures, it must be noted that they 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**.

- **[Waste]** We define waste for an allocation A = $\langle a_i, \ldots, a_T \rangle$ under an available bandwidth vector W to be

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

  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 finds that all M chunks for all future epochs have been pre-fetched. Waste is an indication, typically, that an algorithm has been too conservative in its allocation of early chunks of bandwidth and should have displayed past epochs at a higher quality. The Variability measure, described below, also reacts to wasted capacity, but combines the effects of variance and waste. Hence the separate measure.
- **[Variability]** The Variability of an allocation A = $\langle a_i, \ldots, a_T \rangle$ is defined to be:

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

  Intuitively, a video stream that has near constant quality will be more pleasant to view than one with large swings in quality. However variance can be a misleading measure since a stream with a constant quality of zero will have no variance. 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.

- **[Smoothness]** We define smoothness of an allocation A = $\langle a_1, \ldots, a_T \rangle$ to be:

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

We require this measure since Variability does not capture all of our intuitive notions of smooth quality. 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. To capture this preference, we use smoothness as a measure. Note that smoothness can be quite large: if there are T epochs, smoothness can be as large as $(T-1)M$. Note that an ordered allocation, however, can have a smoothness no greater than M.

4 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 $A = \langle a_1, a_2, \cdots, a_T \rangle$ and prove that this allocation that minimizes waste and variability metrics and has near-minimum smoothness (i.e., no larger than M) 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 $a_n(j) = M$ then 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 proceed to prove results about the optimality of the allocation produced by Best-Allocator. We begin by proving that no other feasible allocation has less waste, then show the allocation has a low smoothness, and last prove, using the theory of majorization, that no other allocation has a lower Variability *V*.

Claim 1 *The allocation formed by Best-Allocator minimizes waste.*

Proof: Consider any bandwidth slot b in epoch i that is not used by Best-Allocator to download a chunk. Clearly, this slot can only be used to play back chunks whose playback epoch is $i+1$ or greater. This bandwidth slot b was considered by Best-Allocator during some iteration j in which k was set to i and the "do nothing" option was selected. This means that $a_n(j) = M$, and it follows from step 3 that Best-Allocator had already allocated chunks for playback of all M layers for all epochs $\ell > k$. In other words, any chunk that bandwidth slot b would be permitted to download (i.e., its playback would occur in a later epoch) had already been scheduled by the jth iteration, and scheduled to a slot that cannot play chunks before or during epoch i. Hence, the only way that an allocation can use b to download a chunk is to "waste" one of the slots that Best-Allocator uses for download. ■

Claim 2 *The allocation $A = \langle a_0, a_1, \cdots, a_T\rangle$ formed by Best-Allocator is non-decreasing, i.e., $a_0 \le a_1 \le \cdots \le a_T$.*

Proof: Assume the claim is false. Then there exists an earliest iteration j where there is an i for which $a_i(j) > a_{i+1}(j)$. Since j is the earliest iteration with such a property, it must be that $a_i(j-1) \le a_{i+1}(j)$. Since a component is incremented by at most one in an iteration, $a_i(j-1) = a_{i+1}(j-1)$. Since the ith component was selected to be incremented in the jth iteration, during step 1, k must have been set to value where $k < i$. Since $a_i(j-1) = a_{i+1}(j-1)$, there is no way that step 3 would set $n = i$, since i cannot be the maximum $l > k$ for which $a_\ell(j)$ is minimal ($\ell = i+1$ would satisfy the minimal property whenever $\ell = i$

did). This is a contradiction. ■

Using this claim, we can now bound the smoothness of the Best-Allocator allocation:

Corollary 4.1 *The smoothness of allocation A formed by Best-Allocator is $a_T - a_0$.*

Clearly, this is not the feasible allocation with minimum smoothness (the allocation $\langle 0, 0, 0 \ldots, 0 \rangle$ has lower smoothness. But no allocation that uses a_T layers in any epoch can have lower smoothness.

Claim 3 *Let $X = \langle x_0, x_1, \cdots, x_{T-1} \rangle$ and $Y = \langle y_0, y_1, \cdots, y_{T-1} \rangle$ be vectors of bandwidth slots where $x_i \leq y_i$ for each i. Then, for a given video, the minimum Variability value V(X) over all feasible allocations under X is no less than the minimum Variability value V(Y) over all feasible allocations under Y.*

Proof: Any allocation that is feasible under X is also feasible under Y. ■

Claim 4 *Let $B = \langle b_1, b_2, \cdots, b_T \rangle$ be any feasible allocation. Let $C = \langle c_1, c_2, \cdots, c_T \rangle$ be the allocation constructed by reordering the components of B such that C is ordered. Then C is also a feasible allocation.*

Proof: (Sketch) Since we are only reordering the components, B and C have identical amounts of waste. To convert B to C, we repeat the following described procedure until it can no longer be performed. Find a pair of components b_i and b_{i+1} (when such a pair exists) where $b_i > b_{i+1}$. Since all the bandwidth slots that are assigned to chunks in epoch i are drawn from epochs $j < i$, any of those slots could be assigned instead to a chunk in epoch $i + 1$. Doing so converts the allocation B to the feasible allocation $\langle b_0, b_1, \cdots, b_{i-1}, b_i - 1, b_{i+1} + 1, b_{i+2}, b_{i+3}, \cdots, b_T \rangle$. By repeating this process as long as such an i exists, we eventually wind up producing an ordered allocation C. Since each time the procedure is applied to a feasible allocation, the resulting allocation is feasible, C - the final allocation generated is also feasible. ■

Definition A vector A = $\langle a_1, a_2, ..., a_T \rangle$ **majorizes** a vector B = $\langle b_1, ..., b_T \rangle$ (written $A \succ B$) if $\sum_{i=1}^{T} a_i = \sum_{i=1}^{T} b_i$ and for all j, $\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} b_i$.

Claim 5 *Let $A = \langle a_1, a_2, \cdots, a_T \rangle$ and $B = \langle b_1, b_2, \cdots, b_T \rangle$ be ordered allocations where $A \succ B$ and $A \neq B$. Then V(A) < V(B).*

Proof: (Sketch) Since A and B are both ordered allocations, we can convert A to B by iterating through a series of allocations $A(0) = A, A_1, A_2, \cdots, A_k = B$ where A_{i+1} is formed from $A_i = \langle \alpha_1, \cdots, \alpha_T \rangle$ by decreasing a component α_i by 1 and increasing another component α_j by 1 where $i < j$ and where $\alpha_i < \alpha_j$ such that A_{i+1} remains ordered and is majorized by A_i. Note that V(A_{i+1}) = V(A_i) +

$(M-(\alpha_i-1))^2+(M-(\alpha_j+1))^2-((M-\alpha_i)^2+(M-(\alpha_j))^2) > V(A_i)$. By repeating this process continually, we eventually convert A to B, and since the resulting vector after each step has a larger Variability than its predecessor, it follows that $V(A) < V(B)$. ■

Claim 6 *Let A be the allocation generated by Best-Allocator and B be any other feasible, ordered allocation with identical waste. Then $A \succ B$.*

Proof: We begin by producing an algorithm that builds B using the following iterative process that constructs ordered allocations $B(0), B(1), B(2), \cdots$ until we eventually produce B: for allocation $B(j)$, take a chunk from the last epoch that is pre-fetched by B but for which a bandwidth slot has not yet been assigned within $B(j-1)$, and assign it to a bandwidth slot in as late an epoch as possible such that $B(j)$ is both feasible and ordered. Letting $A(j)$ be the vector described in the Best-Allocator algorithm, we will show that $A(j) \succ B(j)$ for all j. Since the wastes are equal, A and B are achieved on the same iteration, and the result holds.

Initially, $A(0)$ and $B(0)$ are empty, hence equal, and trivially $A(0) \succ B(0)$. To show $A(j) \succ B(j)$, first consider the case where $A(j-1) = B(j-1)$. Best-Allocator chooses its next chunk for download in the earliest epoch that does not violate the conditions that $A(j)$ is ordered and feasible. In other words, B cannot choose a chunk in an earlier epoch without violating similar conditions, and hence, regardless of the epoch from which $B(j)$ selects its chunk, $A(j) \succ B(j)$.

Finally, assume the claim is false. Then there is some minimum j for which $A(i) \succ B(i)$ for all $i < j$, but where $A(j)$ does not majorize $B(j)$. For this to happen, the epoch of the chunk chosen by B during the jth iteration must be from an earlier epoch than the chunk chosen by A during the jth iteration. In this jth iteration, define t_B to be the epoch chosen by B, and define t_A be the epoch chosen by A, where $t_B < t_A$. Since B selected a chunk in epoch t_B, the bandwidth slot that was assigned to the chunk must be from an epoch earlier than t_B, otherwise B's selection would not be feasible. The fact that A did not select a chunk in front of t_A means that $a_{t_B}(j-1) = a_{t_B+1}(j-1) = \cdots = a_{t_A}(j-1)$. Let h be the value of these components.

Since $a_i(j-1) = a_i(j)$ for all $i < t_A$ and $b_i(j-1) = b_i(j)$ for all $i < t_B$, since $A(j-1) \succ B(j-1)$, and since $A(j)$ does not majorize $B(j)$, it must be the case that $a_i(j-1) = b_i(j-1)$ for all $i < t_B$ such that $\sum_{i=0}^{k} a_i(j) = \sum_{i=0}^{k} b_i(j)$ for all $k < t_B$. Also, since $A(j-1) \succ B(j-1)$ and one and only one component grew at or in front of the t_Ath component in both A and B, we must still have that $\sum_{i=0}^{k} a_i(j) \geq \sum_{i=0}^{k} b_i(j)$ for all $k \geq t_A$. It follows that the smallest value of k that prevents $A(j)$ from majorizing $B(j)$ must have the value $t_B \leq k < t_A$, i.e., there exists $k, t_B \leq k < t_A$ where $\sum_{i=0}^{k} a_i(j) < \sum_{i=0}^{k} b_i(j)$. We call the value of k the *go-ahead position* since it represents the first position of the component where $A(j)$ does not majorize $B(j)$ for the first time.

The proof proceeds via several observations involving this go-ahead position, k:

- Observation 1: $\sum_{i=0}^{k} a_i(j-1) = \sum_{i=0}^{k} b_i(j-1)$. This holds since $\sum_{i=0}^{k} a_i(j) < \sum_{i=0}^{k} b_i(j) = \sum_{i=0}^{k} b_i(j-1) + 1 \leq \sum_{i=0}^{k} a_i(j-1) + 1 = \sum_{i=0}^{k} a_i(j) + 1$, since no component $a_i, i < t_A$ changes value between the $j-1$st and jth rounds.
- Observation 2: $b_k(j-1) \geq h$. If $b_k(j-1) < h$, then $b_k(j) \leq h$. But we must have $b_k(j) > a_k(j)$ for it to be the go-ahead position, but since $t_B \leq k < t_A$, we have that $a_k(j) = h$.
- Observation 3: $b_k(j-1) \leq h$. If $b_k(j-1) > h$, then, since $B(j-1)$ is ordered, we have that $b_{k+1}(j-1) > h$. Since $t_B < k+1 \leq t_A$, we therefore have $a_{k+1}(j-1) = h$, hence $b_{k+1}(j-1) > a_{k+1}(j-1)$. Since, from observation 1, we have that $\sum_{i=0}^{k} a_i(j-1) = \sum_{i=0}^{k} b_i(j-1)$, it follows that $\sum_{i=0}^{k+1} b_i(j-1) > \sum_{i=0}^{k+1} a_i(j-1)$, contradicting $A(j-1) \succ B(j-1)$.

By Observation 3, we have that $k = t_B$. Otherwise, since only the t_Bth component in B changes during the jth iteration, $b_k(j) = b_k(j-1) = h$. Since $B(j)$ is ordered, we must have $b_i(j) \leq b_k(j)$ for all $i < k$, hence $b_i(j) \leq h$ for all $i < k$. Since, $b_i(j) = a_i(j)$ for all $i < t_B$, and all $b_i(j) \leq b_k(j) = h = a_i(j)$ for all $t_B < i \leq k$, we would have $\sum_{i=0}^{k} a_i(j) \geq \sum_{i=0}^{k} b_i(j)$, contradicting k being the go-ahead position.

With $k = t_B$, we must have $b_k(j-1) < b_{k+1}(j-1)$, otherwise the kth component cannot be incremented without violating the ordering property. But Observations 2 and 3 combined give us that $b_k(j-1) = h$, so then $b_{k+1}(j-1) > b_k(j-1) = h = a_{k+1}(j-1)$, which, combined with Observation 1, yields $\sum_{i=0}^{k+1} b_i(j-1) > \sum_{i=0}^{k+1} a_i(j-1)$, contradicting $A(j-1) \succ B(j-1)$. This gives a contradiction. ■

With the four claims above, we can now prove our final result, that Best-Allocator's allocation has is a feasible allocation with minimal Variability:

Claim 7 *Let A be the allocation generated by Best-Allocator and let B be any other feasible (not necessarily ordered) allocation with identical waste. Then V(A) ≤ V(B).*

Proof: By Claim 4, by reordering B to produce ordered allocation C, C must be a feasible allocation. In addition, V(B) = V(C) since the order of components does not affect the value of the Variability metric. Claim 6 gives us that A majorizes C, and Claim 5 gives us therefore that V(A) ≤ V(C) (note the possible equality because C may equal A). ■

Claim 8 *Let A be the allocation generated by Best-Allocator and let B be any other feasible (not necessarily ordered) allocation with possibly different waste. Then V(A) ≤ V(B).*

Proof: Let $Y = \langle y_0, y_1, \cdots, y_{T-1} \rangle$ be the original vector of bandwidth slots, and let $X = \langle x_0, x_1, \cdots, x_{T-1} \rangle$ be a vector of bandwidth slots that could be used to

feasibly achieve allocation B where every slot is used to achieve B and where $x_i \leq y_i$ for all i. Note that such a vector can easily be formed by taking Y, constructing allocation B, and removing any unused slots from the vector of bandwidth slots. Since every slot of X is used to construct B, B is an allocation of minimal waste under vector X. Let A' be the allocation constructed by Best-Allocator under X. By Claim 1, A' also has minimal waste and hence A' and B pre-fetch the same number of chunks. By Claim 7, $V(A') \leq V(B)$. Then by Claim 3, the minimum Variability value over all feasible allocations under X is larger than the minimum Variability value over all feasible allocations under Y, i.e., $V(A') \geq V(A)$. Since we have $V(A) \leq V(A') \leq V(B)$, the result is proved. ■

5 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 coded videos in the context of the model described in Section 3. The off-line algorithm "Best-Allocation" presented in Sec 4 achieves the best possible performance for the given input by making its allocation decisions *after* receiving as input the number of bandwidth-slots that can be downloaded in each epoch. In reality, we are not given an advance indication of what the future bandwidth delivery rate will be. Hence, these on-line algorithms must make their decision without knowing the future bandwidth availability.

We first look at the tradeoffs involved in the decision making process at the client. 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.

5.1 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 nearest future epoch for which it has not yet downloaded all the chunks. With such an allocation, the variance in the quality of the video will be as large as the variance of the bandwidth to the servers, when the mean of the

input is small relative to the capacity of each bin. As the mean increases, it will produce smoother allocations. 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. In case of ties it chooses the earliest epoch. Such an allocation strategy will have the effect of downloading **all** of layer-1 of the video stream, then downloading all of layer-2 and so on. This approach will produce unchanging and smooth quality, but will waste capacity and the overall number of layers viewed will be small. Consider the situation where we have a constant bandwidth of $m_t = M$ for all epochs. Even in such a favorable bandwidth environment, with the Smallest-Bin allocator, we would watch the video at full quality only for the last $1/M$ fraction of the video, and we would watch the only a single layer of the video for the first $1/M$ fraction.

5.2 Constrained allocators

Intuitively we would like algorithms that operate somewhere between the two extremes of the same-index and smallest-bin allocators. Our approach to this problem is to design algorithms that attempt to maximize the current quality, but with smoothness constraints based on perceptual quality studies [1] that indicate that users dislike quick changes in video quality, and are particularly sensitive to *decreases* in quality. We therefore designed a suite of online algorithms with constraints on the allowed changes in quality. In particular we constrain the **downhill slope** of an allocation; i.e. we specify the maximum allowable downhill slope an allocation can have. Within this restriction the different online algorithm in our suite attempt to maximize different desirable qualities all the while ensuring that the maximum downhill slope remains within the bound.

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 flavors of these 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 maxi-

mize 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. The allocation of this algorithm in response to the input $\langle 6, 7, 9, 11 \rangle$ and a constraint $C = 1$ is shown in Fig. 2(A). [3]

- **[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 excess current bandwidth is an aberration which will be compensated by drops in the bandwidth in the future and therefore current excess bandwidth should be used to download future parts of the video. The allocation of the Mean-Hill algorithm with a slope-constraint of $C = 1$ in response to the input $\langle 6, 7, 9, 11 \rangle$ is shown in Fig. 2(B) when $\mu_w = 3$. It can be seen how it downloads future chunks of the video, once bandwidth exceeds the mean.
- **[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. The allocation of this Wide-Hill algorithm with a slope-constraint of $C = 1$ in response to the input $\langle 6, 7, 9, 11 \rangle$ is shown in Fig. 2(C) when $\mu_w = 3$. Its proclivity to build allocations up to to the mean, then grow into the future while maintaining the slope constraint can clearly be seen.

Fig. 2 typifies the allocations of the three algorithms when the slope-constraint $C = 1$. The example uses the 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.

[3] All our methodology, analysis, simulation and experiments pertain to variable bit rate videos. The epoch-lengths vary to account for the variability in the bit-rates of videos resulting in constant-chunk sizes. The constant chunk-size eases analysis, but should not be mistaken for constant bit-rate video. Further, to simplify visualization, Fig. 2 shows constant epoch-lengths. Again this does not imply that constant bit-rate videos were considered.

6 Results

In this section we present and evaluate the performance of the off-line Best-Allocation 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-Allocation to minimize Variability [4] , smoothness and waste.

In the first simulation 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 condition: 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 simulation 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. Such a study will tell us if our algorithms function in real-life bandwidth conditions, and furthermore give us an indication about the optimal decision-interval, and if such choices are critical. As we increase the average epoch-length over which we compute the network bandwidth, as expected, the variance of the bandwidth steadily falls. Therefore our chart plots performance as a function of variance.

6.1 Experimental Setup

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 a one-second sampling of $w(t)$ was achieved by tracking the number of bytes of a video that was downloaded each second. The two traces chosen 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. A bin-size M of 6 was used throughout the experiments.

[4] 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 $chunks^2$.

To study performance as a function of the mean of the input as it approaches the bin-capacity, we independently generate inputs using a Uniform distribution, varying the mean of the distribution over the various experiments (and thereby the varying the standard-deviations well) to the bin-capacity. We then plot the performance measures as a function of the mean as illustrated in Fig. 3.

Secondly we study the performance of the different algorithms as the variance of the input increases. If different algorithms provide different performances under different mean and variance conditions, such an analysis will allow us to choose the optimal strategy as a functions of the input's mean and variance. The inputs in this experiment are generated independently from a Gaussian distribution. The mean is held constant to half the bin-capacity and the standard-deviation is increased gradually. Again, the variability, the smoothness and waste are measured as functions of standard-deviation as illustrated in Fig. 4.

6.2.1 Analysis of Simulations

Here we provide a synopsis of the results as observed from the graphs of Fig. 3 and Fig. 4.

- The constrained-hill algorithms (Largest-Hill, Mean-Hill and Wide-Hill) vastly outperform the naive allocators (Smallest-Bin and Same-Index) with regard to Variability. The Smallest-Bin algorithm produces a smooth allocation as expected but performs poorly on all other measures. The Same-Index allocator wastes almost no capacity but produces non-smooth allocations.
- The constrained-hill algorithms perform close to the bound provided by the offline Best-allocation. While the Best allocator wastes almost no chunks in all situations, the constrained-hill algorithms waste about 1 percent of the chunks under moderate conditions and 2-5 percent under extreme strain.
- The Same-Index allocator produces less-smooth fillings as the variance goes up, but produces more smooth fillings as the mean increases. This is because its variance is the same as the inputs, but capped by the capacity of the bins.
- Of the constrained-hill algorithms, the Wide-Hill allocator produces a smoother packing but wastes more inputs. The Mean-Hill allocator produces small waste and good smoothness.
- The Mean-Hill algorithm performs marginally better than the Largest-Hill algorithm for the case of the Uniform distribution where the mean and variance increase in tandem, and the mean approaches the bin-capacity. The Wide-Hill tends to waste a slightly larger amount of the input under milder loads.
- The Largest-Hill algorithm wastes fewer inputs, but produces a less smooth packing than the Mean-Hill and Wide-Hill algorithms for Gaussian inputs where the

standard-deviation approaches the mean, and the mean is held to half the bin-capacity.

The Mean-Hill algorithm is marginally more robust than the Largest-Hill and Wide-Hill algorithm when the mean of the inputs approach the bin-capacity, while they all produce similar performances when the mean is half the capacity and the standard-deviation approaches the mean. The Largest-Hill algorithm wastes less as the standard deviation increases, while producing a less smooth allocation. This is probably because the Mean-Hill algorithm is more likely to allocate chunks to the future, and when the standard-deviation is high, allocating chunks to the future is more likely to produce wastes due to spikes in the last epochs of the input. The Mean-Hill algorithm always produces a smoother packing than the Largest-Hill, as it tends to even-out larger hills.

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

6.3 Bandwidth Traces

In this section we present the results of applying the algorithms to real data. Two sets of traces were obtained, the first on a T1 network, and the second through a DSL line. We created a program based on modifying the Limewire Open source code [26] 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 heuristic algorithms when applied to these traces for an epoch-length of 1 second. We repeated the experiment for increasing epoch-lengths of 2,4,8,16,32 and 128 seconds, to study the effect of time-scales on the performance of the heuristic algorithms.

The bandwidths of the two traces for the first 100 epochs is plotted in Fig. 5. It can be seen that there is large bandwidth variation across all scales. The performance of the heuristic algorithms on these two traces are charted below.

6.3.1 T1 connection

This trace was run on a computer connected to the Internet through a T1 connection. The trace lasted for 3,663 seconds and downloaded a 148 MB video. As many as 5 servers were serving the video simultaneously. The algorithms were run on this trace for different average epoch lengths of 1, 2, 4, 8, 16, 32, 64 and 128 seconds. The standard deviation of the bandwidth reduced from 1.46 to 1.11 in response.

The performances are charted in Fig. 6 as a function of these different standard-deviations resulting from varying the average epoch-lengths.

6.3.2 DSL Connection

The second 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 as previously and the standard deviation of the bandwidth reduced from 2.97 to 1.64 in response. The performances are charted in Fig. 7 as a function of these different standard-deviations resulting from varying the average epoch-lengths.

6.3.3 Analysis of Trace Results

- With both the T1 connection and the DSL connection, the Mean-Hill and Wide-Hill perform close to the bound provided by the best-allocator. The Largest-Hill algorithm lags in performance and starts diverging as variance increases.
- The Mean-Hill and the Wide-Hill algorithms show consistent, and sometimes even improving performance as the variance increases (which correlates to smaller epoch-lengths). The extra variance, when epoch lengths are small, is probably compensated by the more accurate bandwidth information used, and the more frequent decision-making.
- The naive algorithms perform relatively poorly on the Variability measure. The Same-index algorithm wastes little input as expected but produces non-smooth fillings, while the Smallest-Bin allocator produces smooth fillings but wastes the most capacity.
- The Mean-Hill and Widest-Hill algorithms perform at near optimal levels throughout and waste less than 1% of the input, with very smooth fillings.

In summary the Mean-Hill and Wide-Hill algorithms provide near-optimal performance in real-life bandwidth scenarios. Furthermore they perform consistently across varying epoch-lengths.

7 Issues

We briefly mention some further issues for consideration and directions of future work here.

First we consider fast-forwards and rewinds. Our approach handles rewinds well,

as all the chunks that correspond to past segments of the video have been downloaded and can be viewed frame-by-frame in the reverse direction. We just need to make sure that we do wait for a time period of k seconds before we discard downloads, where k is the amount of time for which rewind is required. Algorithms like the Mean-Hill algorithm lend themselves well to the support of fast-forwarding as they use any bandwidth in excess of the mean to download future segments of the video, one ayer at a time. After and during a fast-forward the viewer may be able to view the video at a lower quality until the algorithms reschedule and download the chunks necessary for normal quality downloads.

A couple of issues that warrant consideration in future research are the problem of scalable-codings where the different ayers are of unequal sizes. It is currently not obvious how such unequal sized chunks can be scheduled. A second interesting question concerns the amount of time spent pre-fetching before the user begins viewing the video. In our model, this time is determined by the length of the zeroth epoch. It may be worthwhile investigating how this time may be reduced, and the effects of such a reduction on performance.

In future work we plan to focus energies on implementing our algorithms and producing real systems that can be used to download and stream scalably-coded coded video. We would like to compare the performance of such a system against other video-streaming algorithms and verify that our algorithms do indeed produce the most even quality under the constraints.

8 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 any one server is willing to devote to an upload. Scalably Coded video, a attractive solution to this problem is gaining popularity as it distributes the user's bandwidth requirement across many peers. We show that in such a scenario, because the user's 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, then present on-line algorithms that vastly outperform naive schedulers. Through simulations we show that our solutions perform close to the best possible performance.

References

[1] M. Zink, O. Kunzel, J. Schmitt, R. Steinmetz, Subjective impression of variations in layer encoded videos, in: International Workshop on Quality of Service, Berkeley, CA,

USA, 2003, pp. 137–154.

[2] R. K. Rajendran, D. Rubenstein, Optimizing the quality of scalable video streams on p2p networks, in: SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems, ACM Press, New York, NY, USA, 2004, pp. 396–397.

[3] R. K. Rajendran, D. Rubenstein, Optimizing the quality of scalable video streams on p2p networks, in: Globecom 2004, 47th annual IEEE Global Telecommunications Conference, November 29th-December 3rd, 2004 - Dallas, USA, 2004.

[4] P. de Cuetos, K. W. Ross, Adaptive rate control for streaming stored rine-grained scalable video, in: NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video, ACM Press, New York, NY, USA, 2002, pp. 3–12.

[5] D. C. Saparilla, Broadcasting and streaming stored video, Ph.D. thesis, supervisor-Keith W. Ross (2000).

[6] D. Saparilla, K. W. Ross, Streaming stored continuous media over fair-share bandwidth, in: NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video, ACM Press, New York, NY, USA, 2002.

[7] T. Kim, M. H. Ammar, A comparison of layering and stream replication video multicast schemes, in: NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video, ACM Press, New York, NY, USA, 2001, pp. 63–72.

[8] J. Taal, J. Pouwelse, R. Lagendijk, Scalable Multiple Description Coding for VIdeo Distribution in P2P networks, in: 24th Picture Coding Symposium, San Francisco, CA, 2004.

[9] Y. Zhao, D. Eager, M. Vernon, Network Bandwidth Requirements for Scalable On-Demand Streaming, in: Proceedings of IEEE INFOCOM'02, New York, NY, 2002.

[10] A. Mahanti, D. L. Eager, M. K. Vernon, D. J. Sundaram-Stukel, Scalable on-demand media streaming with packet loss recovery, IEEE/ACM Trans. Netw. 11 (2) (2003) 195–209.

[11] H. Tan, D. Eager, 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, 2002.

[12] J. E. G. Coffman, P. Momcilovic, P. Jelenkovic, The dyadic stream merging algorithm, J. Algorithms 43 (1) (2002) 120–137.

[13] V. N. Padmanabhan, H. J. Wang, P. A. Chou, K. Sripanidkulchai, Distributing streaming media content using cooperative networking, in: NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video, ACM Press, New York, NY, USA, 2002, pp. 177–186.

[14] R. Rejaie, M. Handley, D. Estrin, Quality adaptation for congestion controlled video playback over the internet, in: SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, ACM Press, New York, NY, USA, 1999, pp. 189–200.

[15] The BitTorrent System, available from http://www.bittorrent.com.

[16] The Coolstreaming System, available from http://www.coolstreaming.org.

[17] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications, in: Proceedings of ACM SIGCOMM'01, San Diego, CA, 2001.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A Scalable Content-Addressable Network, in: Proceedings of ACM SIGCOMM'01, San Diego, CA, 2001.

[19] B. Zhao, J. Kubiatowicz, A. Joseph, Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing, Tech. rep., UC Berkeley (April 2001).

[20] A. Rowstron, 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, 2001.

[21] The Gnutella Protocol Specification v0.4, revision 1.2, available from http://gnutella.wego.com.

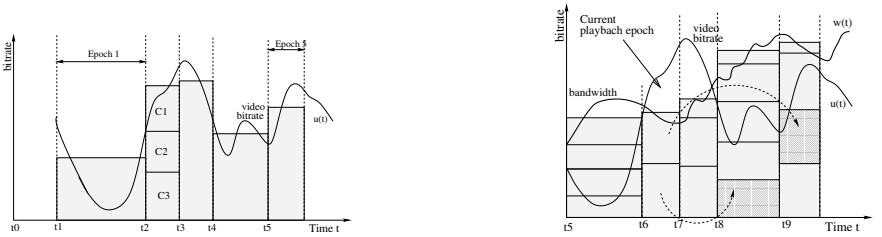
[22] Overview of the mpeg-4 standard, available at http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm (2002).
URL
http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm

[23] H. Radha, M. van der Schaar, Y. Chen, The mpeg-4 fine-grained scalable video coding method for multimedia streaming over ip, in: IEEE Trans. on Multimedia, 2001.

[24] P. Rodriguez, E. Biersack, Parallel-Access for Mirror Sites in the Internet, in: Proceedings of IEEE INFOCOM'00, Tel-Aviv, Israel, 2000.

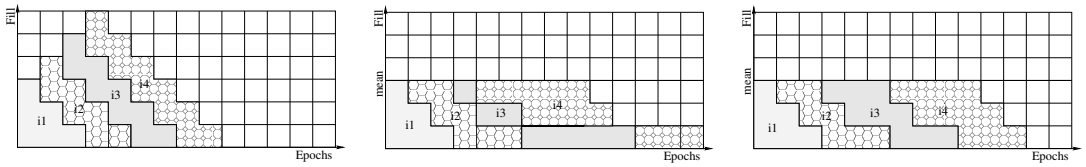
[25] J. Byers, M. Luby, M. Mitzenmacher, Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads, in: Proceedings of IEEE INFOCOM'99, New York, NY, 1999.

[26] Limewire open-source software, available from http://www.limewire.org/project/www/Docs.html.
URL http://www.limewire.org/project/www/Docs.html



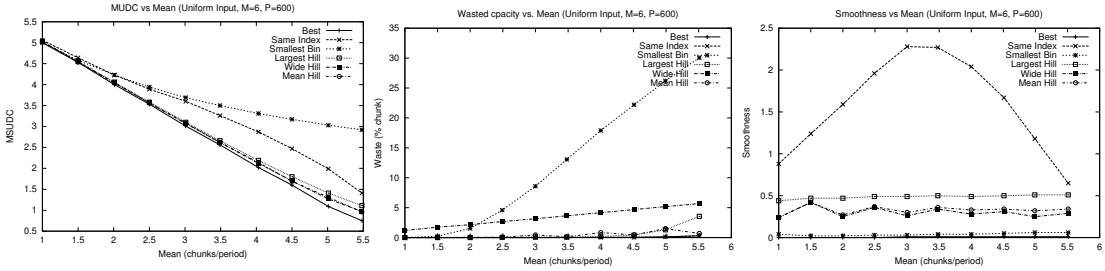
(A)Formation of chunks (B)Pre-fetching of chunks

Fig. 1. The Model



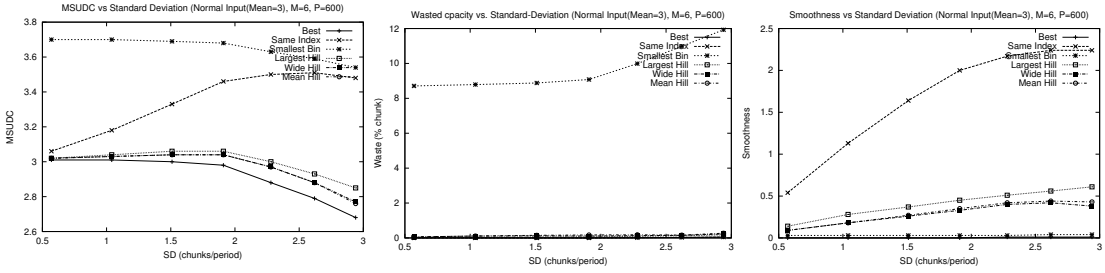
(A) Largest-Hill (B) Mean-Hill (C) Wide-Hill

Fig. 2. Three slope-constrained algorithms



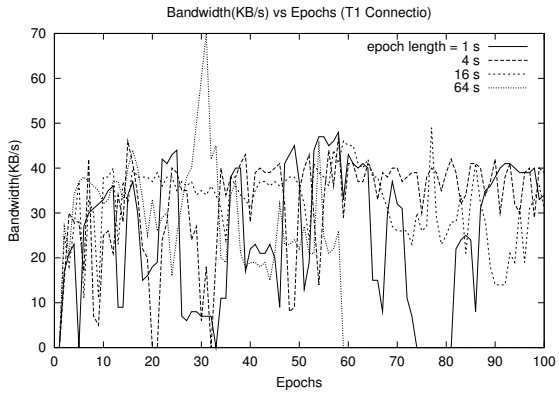
Variability % Waste Smoothness

Fig. 3. Performance vs Uniform Input's Mean (P=600)

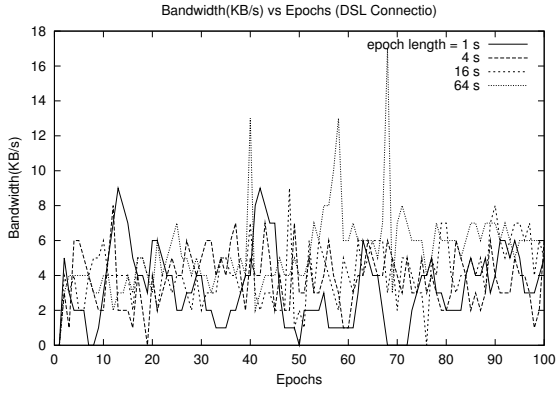


Variability % Waste Smoothness

Fig. 4. Performance vs Gaussian Input's Variance (P=600)

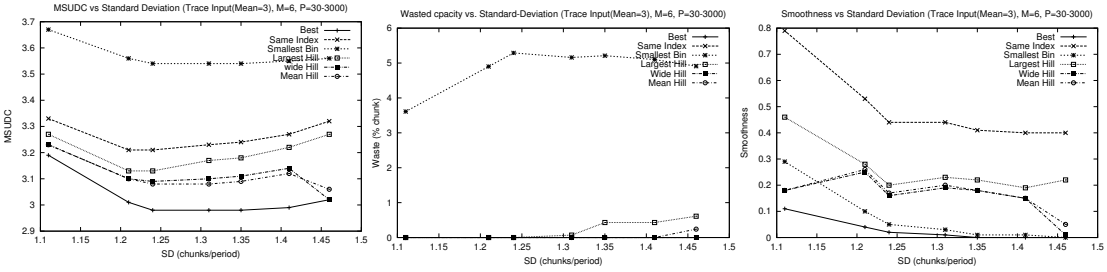


T1 trace



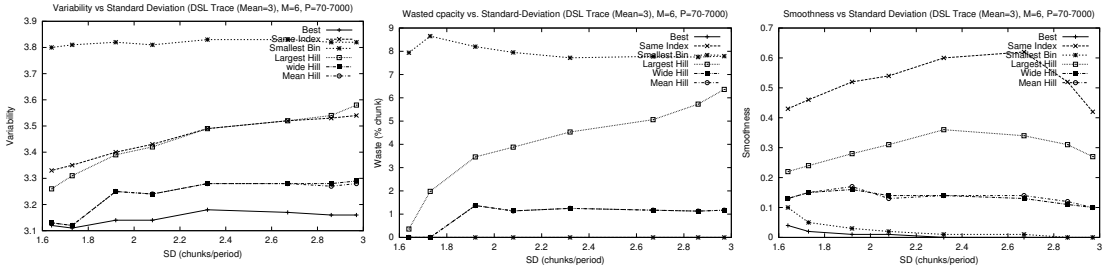
DSL trace

Fig. 5. Bandwidth Traces at different time-scales



Variability % Waste Smoothness

Fig. 6. Performance vs T1 Trace Input's Variance (P=30-3000)



Variability % Waste Smoothness

Fig. 7. Performance vs DSL Trace Input's Variance (P=100-10000)