NEAR-OPTIMAL HEURISTIC SOLUTIONS FOR TRUNCATED HARMONIC WINDOWS SCHEDULING AND HARMONIC GROUP WINDOWS SCHEDULING

by

Zhiwen Lin

B.Sc., University of Science and Technology of China, 1987

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the School
of
Computing Science

SIMON FRASER UNIVERSITY

Fall 2004

Approval

Name:	**Zhiwen Lin**
Degree:	**Master of Science**
Title of Thesis:	**Near-Optimal Heuristic Solutions for Truncated Harmonic Windows Scheduling and Harmonic Group Windows Scheduling**

Examining Committee:

Chair: **Dr. Andrei Bulatov**
Assistant Professor of Computing Science

Dr. Tiko Kameda
Senior Supervisor
Professor of Computing Science

Dr. Ramesh Krishnamurti
Supervisor
Professor of Computing Science

Dr. Jiangchuan Liu
Examiner
Assistant Professor of Computing Science

Date Approved: October 12, 2004

Abstract

Dividing a video into many segments and then broadcasting each segment periodically has proved to be an efficient and cost-effective way of providing near Video-on-Demand services. Some of the known broadcasting schemes, such as Fixed-Delay Pagoda Broadcasting (FDPB), adopt the fixed-delay policy, which requires the user to wait for a fixed time before watching a video. Our first broadcasting scheme, the *Generalized Fixed-Delay Pagoda Broadcasting* (*GFDPB*), based on the fixed-delay policy, improves Bar-Noy et al.'s greedy algorithm for the Harmonic Windows Scheduling Problem. GFDPB achieves the lowest maximum waiting time among all the known protocols using segments of equal duration and channels of equal bandwidth. In addition, its performance is very close to the theoretical optimum. Second, we define the *Harmonic Group Windows Scheduling* (*HGWS*) problem and present a new broadcasting scheme to solve it, *Harmonic Page-set Broadcasting* (*HPB*), which provides the lowest average waiting time of all currently known protocols by using the fewest channels for given server bandwidth. Finally, we present a hybrid broadcasting scheme, *Preloading Page-Set Broadcasting* (*PPSB*), which compromises between the average waiting time and the maximum waiting time of HPB. While still providing the shortest average waiting time of all known protocols using segments of equal duration and channels of equal bandwidth, PPSB achieves much shorter maximum waiting time than HPB. Furthermore, PPSB provides a very desirable trade-off between the average waiting time and the maximum waiting for a given server bandwidth, while guaranteeing that its maximum waiting time is only 1/3 longer than its average waiting time.

Dedication

献给我亲爱的母亲 — 杨瑞珍

To my dear mother,   Runzhen Yang

Acknowledgements

I would like to express my deepest gratitude to my senior supervisor, Dr. Tiko Kameda, for his guidance and continuous support throughout the duration of my study and research. His serious attitude toward research impresses me most deeply and sets an excellent example for me. My gratitude also goes to my supervisor Dr. Ramesh Krishnamurti and the examiner, Dr. Jiangchuan Liu, for their valuable comments and advice on my thesis.

Special thanks are reserved for Yi Sun for his discussion with me about the Harmonic Group Windows Scheduling Problem. I am also thankful to other members of the Distributed Computing Laboratory, Zhong Zhang and Shufang Wu, for their help and friendship.

Finally, I am especially grateful to my parents for their love, support and encouragement over the years. I am equally grateful to my wife Hong Huang for her understanding and support during my study.

Table of Contents

List of Figures

List of Tables

Chapter One
Introduction

With the advancement of broadband networking technology and the increasing power of computers, video-on-demand (VOD) has appeared as an important technology for many multimedia applications such as news on demand, digital libraries, home entertainment, and distance learning. A common VOD service allows remote users to play back any video from a large collection stored on one or more servers at the time of their choice.

1.1 VOD System Architecture

A typical video-on-demand architecture [50] consists of three critical subsystems: video servers, high speed wide-area and/or local distribution networks, and user populations (see Figure 1.1 for a single user population). A high capacity video server can support a large number of video streams and deliver high quality digitized video data to clients either over a local LAN or remotely over high speed network connections. In addition to providing buffers for video segments periodically delivered from the video servers, a set-top box, along with a television monitor and a remote control, enables viewers to be connected to a video source (video server); viewers browse through a selection of videos, and then receive and display the selected video. Each local video server cluster is dedicated to a single user population. It may store a complete or partial set of videos from the video collection, work as a proxy server, and/or administer admission control before accepting new requests. A remote video server cluster may also be archival in nature providing a permanent repository for all videos. The most popular videos are replicated and stored on different servers in a local cluster. The service from the local cluster is provided over a LAN such as an ATM LAN, a hybrid fiber coaxial (HFC) network, or a Gigabit Ethernet network.

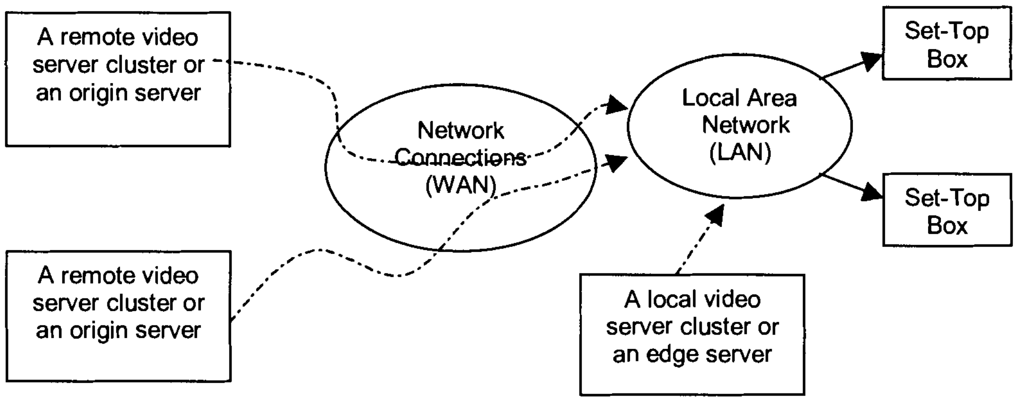

A remote video server cluster or an origin server
Network Connections (WAN)
Local Area Network (LAN)
Set-Top Box
Set-Top Box
A remote video server cluster or an origin server
A local video server cluster or an edge server

Figure 1.1 Elements of a VOD system

The recently published HP open VOD solution for cable [12] uses Gigabit Ethernet to interconnect with video server farms in regional headends where video streams are formatted and organized into LANs, and then distributes VOD streams to individual subscribers across the existing hybrid fiber coax (HFC) infrastructure. The video delivery and management portion of the solution is Kasenna's MediaBase XMP [13]. Small servers are deployed in a distributed server architecture based on the "origin server" and the "edge server" (see Figure 1.1) models. The small servers work as both proxies and part of servers and are deployed "closer" to end users to address quality, bandwidth, and scalability constraints.

1.2 VOD Protocols

1.2.1 Unicast Scheme

The VOD delivery solution currently used by hotels and cable TV companies initiates a dedicated channel for each client's request and provides all VCR-like control such as forward, rewind, pause, and search [2, 3]. Since a video is an isochronous medium, a video server has to reserve a sufficient amount of network bandwidth and input-output bandwidth for each video stream before committing to a client's request [4]. Under a heavy load, the server bandwidth quickly becomes a bottleneck, because the growth of the number of channels may never be able to keep up with the growth in the number of clients. Therefore, this solution does not scale well and the number of clients that can be served simultaneously is limited.

The Personal Video Record (PVR) service [9] is another recent service. It makes a set-top box receive and record selected TV programs for late "on-demand" viewing at one's leisure with full VCR-like control; also, it provides VCR-like control over live TV broadcasts, where users can pause and rewind the current broadcast. The set-top box contains an encoder and a hard disk drive to store recorded programming. Most significantly, this set-top box PVR makes no bandwidth demand on the broadband service provider's network. However, it is not really a VOD service and indeed it is just a TV program recorder. Furthermore, it has many limitations on how much programming it can store and how many videos it can record simultaneously (presently at most two); it has a risk of hard disk drive failure and a set-top box can only server one TV at a time.

The limitations of set-top box PVR are resolved by network PVR [9]. Network PVR offers users essentially the same functionality as set-top box PVR except that disk storages, encoders, and software intelligence are located on video servers in regional headends or digital hubs. The network PVR service doesn't require hard disk drives in consumer homes, but, cable operators must allocate bandwidth to accommodate the on-demand video streams requested from the server. This again leads to the scalability problem mentioned above.

Many altematives known as near-VOD services have been proposed to tackle the scalability problem by sacrificing some VCR functions. The schemes described below in this chapter belong to this category.

One technique to reduce an individual server's load, such as proxy caching [5], [6], [7], [8], is to cache streams at various nodes in a network. A proxy is a distributed caching scheme to enhance the scalability of web services over the Internet. By placing proxies on the edge of a backbone network, the network traffic and server load are substantially reduced, as it is often the case that 80 percent of demands are for a few (10 to 20) of the most popular videos [10] [11]. In the absence of proxy caching, a popular video is transferred through the same network link once per request. This results in server overload, network congestion, higher latency, and even the possibility of the rejection of a client's request as mentioned earlier. Furthermore, proxies can deliver their contents over unicast networks and do away with client buffers.

The limitation of proxy caching is that the temporal distances of the aggregated client requests cannot exceed the free buffer size (a temporal distance is defined as the interval between any two requests in [6]). If a proxy buffer is not large enough to cache an entire movie and the temporal distance between the present request and the earliest online request exceeds the free buffer space, the proxy server still needs to initiate a unicast server stream for each request [6]. Although [8] has proposed a cooperative proxy scheme that dynamically adjusts buffer allocation to give a priority to aggregate the requests with short temporal distances and decrease the number of unicast server streams, there are still many repeated unicast server streams between each video server and a proxy. During peak hours, unicast client streams from each proxy to client cost much more bandwidth than multicast schemes or broadcast schemes. Moreover, each proxy can only support a small number of clients because of the limitation of each proxy's outward bandwidth and storage space; therefore, many proxies are required in a VOD system. Another similar technique is to replicate servers with the same content or the same popular movie content at various sites in the network. Both of these approaches attempt to achieve better scaling by decentralizing the location of a media object, but they drive up the cost of providing VOD service substantially.

1.2.2 Non-Periodic Multicast Scheme

Multicast is exploited to reduce the number of streams required by a server to support a given number of clients. In multicast, a number of requests are grouped together according to some scheduling policies and served by a video stream.

The *batching* approach was originally proposed in [14]. The basic idea is to delay requests for different videos for a certain amount of time so more requests for the same video arriving during the current batching interval may be serviced by the same stream. The system has limited bandwidth as well as a limited number of available channels, and there may be a large number of requests for different video at any time; some requests are for popular movies and some are for less popular movies. Therefore, an efficient scheduling policy based on the arrival of requests is required to efficiently utilize channels [15, 16, 17, 18, 19, 20].

Two common scheduling policies, first-come-first-served (FCFS) and maximum queue length (MQL), are studied in [15]. Under the FCFS policy, requests for all movies join a single queue. Once a multicast channel is available or after a batching interval, the client at the front of the queue is served as well as all the requests in the queue for the same movie. The FCFS policy seems like a fair policy since it selects a movie request independent of its popularity ("hot" or "cold"). Under the MQL policy, requests for each movie join a separate queue, and the movie with the maximum queue length is selected for multicast. One drawback of this policy is that it may choose only hot movies, since there are very few requests for cold movies within a short time period. However, MQL can better utilize a small server capacity to reduce the overall reneging probability (the probability that clients may cancel their requests because of long waits). It is shown in [15] that since the MQL policy does not take into account client waiting, the MQL policy may not even perform as well as the FCFS policy in terms of reneging probability if the reneging probability of a client depends on the amount of waiting. From this point of view, the FCFS policy is the preferred policy.

The *maximum factored queue length* (MFQ) policy, studied in [17], is an improvement to the MQL policy. Under the MFQ policy, whenever a channel becomes available, the video with the largest value of $q_i * \Delta t_i$ is scheduled, where q_i represents the length of the queue for video i, and Δt_i is the interval since the last time video i was scheduled. If a movie has not been scheduled for a long period time, its priority will increase accordingly. Therefore, the MFQ policy guarantees that the starvation of cold movies will never occur. In [17], it is shown that the MFQ policy outperforms both the FCFS and the MQL policy in terms of the average waiting time if there is no reneging, or in terms of reneging probability if reneging exists. The MFQ policy also performs better than the MQL policy and is nearly comparable to the FCFS policy in terms of fairness.

The question of when to launch a new channel best if a channel is available in a batching approach was studied in [20]. The authors analyze a number of batching schemes and compare client waiting times and system profitability (the number of current channels and the number of clients per channel). The first scheme studied in [20] is a window-based scheme in which clients are batched for a fixed period of time W before they are served, so client delay is bounded by a maximum value W. Since system profitability depends on the average number of clients in a batch, window-based schemes have good profits if the arrival rate is high, but have poor profits if the arrival rate is low. The second scheme is a batch-size based scheme in which the profitability is maintained by launching a new channel whenever a certain number *M* of clients are collected in a batch. However, if the request rate increases, the average number of channels can grow unbounded. To combine the advantage and lessen the disadvantage of the above two schemes, they proposed a third scheme, a self-adaptive scheme, with three parameters: 1) a minimum window size, W_{min} in which as many clients as possible are batched if the arrival rate is high; 2) a batch-size *M* which is used to guarantee a profitability if the arrival rate is not so high; 3) a maximum windows size, W_{max} which is used to bound client delay if the arrival rate is low. The last scheme they considered is a moving-average scheme in which they force the mean delay of all clients in a batch to be equal to a certain average user delay requirement.

As to the question of how to place a large amount of video contents on a set of video servers to optimize the performance of a batching VOD system, a recently published paper [33] proposes an optimal video placement scheme. The video placement problem is formulated as a modified bin-packing problem which can be effectively solved by the hybrid generic approach proposed first in [32]. Given a specified blocking probability, the minimum batching interval is derived and the corresponding video file placement is obtained while the server capacity usage is minimized by the scheme.

The major drawback of batching schemes is that each client has to wait for a batching interval until the request is served and yet the batching interval cannot be too short to benefit from multicasting. Thus, clients making early requests are likely to renege if they are kept waiting too long. To overcome this problem, a technique referred to as a *patching* scheme [22, 23, 24, 25] or a *stream tapping* scheme [36, 37] is proposed on the top of the batching method. Patching allows a batch of new incoming requests to join an on-going multicast and take advantage of the same data channel. In this case, video requests are first served by a patching (multicast or unicast) channel for the missed portion of a video while simultaneously buffering the rest portion coming from the existing multicast. The existing multicast is called *regular* channel that multicasts the whole video. The patching channel plays a video until the starting point of the buffered portion of the video, then, the patching channel is released and the client plays from his buffer while buffering the future video data from the regular channel. In this way, service latency is shortened or eliminated without compromising the benefit of the multicast.

Since the workload of a patching channel increases as the age of the latest regular channel increases, it may be more efficient to start a new regular channel rather than continue patching the latest regular channel when the regular channel reaches a certain age. [25] presents a proof of this point for VOD patching without service latency, and proposes a method to determine the optimal *patching window*, and calls this scheme optimized patching. In this scheme, a patching window is a time period after the launch of a regular channel during which patching channels are used. After every patching window interval, a regular channel is launched to maximize the data sharing. Recently, a new technique called a *two-level patching* scheme has been proposed [28], and the scheme introduces two level patching channels to minimize the server bandwidth requirement. Unlike a patching scheme where clients receive streams from at most two determined channels, clients in the two-level patching scheme receive streams from at most three determined channels rather than two. It is shown in [28] that the two-level patching scheme has a significantly lower bandwidth requirement than the optimized patching scheme.

Stream merge schemes [26, 27, 29, 30, 31] belong to another multicast approach and are hierarchical dynamic multicast schemes. In stream merge schemes, clients receive streams from normally at most two channels at the same time. Later streams are repeatedly merged into the former streams for the same movie, leading to a hierarchical merging (or patching) structure. The optimal merge trees are also studied in the above papers. All the above multicast schemes except [30, 31] focus on immediate service. However, [30, 31] study the delay guaranteed system, in which time is divided into intervals of unit length. A new stream is launched at the end of an interval if there is at least one request in this interval, or no new streams are launched if there are no new requests in the interval. It is shown in [30] that the delay guaranteed on-line algorithm is much simpler than any immediate service stream merging algorithm, and performs well in terms of total server bandwidth usage when the mean inter-arrival time of clients is less than the guaranteed start-up delay.

Another interesting approach is the dynamic skyscraper technique [34, 35]. Based on the FCFS policy, the approach uses a set of static skyscraper broadcast channels, which is introduced in the next section, to broadcast different videos according to recently batched requests. Non-overlapping clusters of skyscraper broadcast periods are identified, and each cluster can broadcast one video continuously, completely and independently. The clusters can then be dynamically scheduled for different videos. If the largest segment duration in a skyscraper broadcast is equal to W slots given K channels, then each new cluster will begin on channel 0 precisely W slots after the beginning of the earliest period in the previous cluster, and the latency in the system is fixed and bounded. Some optimization methods, such as new segment size progressions and channel stealing, are also studied to address the problem of unused channel bandwidth.

The other related solutions are piggyback schemes [38, 39, 40, 41]. Piggyback schemes dynamically speed up and slow down client display rates in order to bring different streams to the same file position, at which time the streams can be merged. However, the maximum rate at which clients can be merged is bounded by the variation in viewing rate (typically 5%) which can be tolerated by a client.

Although some multicast schemes can provide bandwidth savings and introduce zero startup delay such as immediate patching, they require more complicated control systems and are not as suitable for high request bursts at peak hours.

1.2.3 Broadcast Scheme

As mentioned in Section 1.2.1, it is reported that 55% to 80% of the overall demands for videos are on a few (10 or 20) very popular videos. Broadcast protocols were introduced to efficiently distribute the top ten to twenty videos to provide near-VOD service where servers use multiple dedicated channels to broadcast a video cooperatively and repetitively. The common idea in these protocols is that the data for each video is divided into fragments or segments and these fragments/segments are broadcast during predefined periods on a set of channels. Clients must be able to receive two or more channels simultaneously and must be able to buffer a fragment/segment that is received before needed for playback. All of these solutions assume a static allocation of bandwidth per transmission; thus, bandwidth savings are achieved only when client request rates are high.

A distinct advantage of this approach is that it can serve a very large community of users by using minimal server bandwidth. In fact, the bandwidth requirement is independent of the number of clients in the system. This makes it scale up extremely well. However, broadcast schemes cannot provide an immediate service, and clients have to wait until the beginning of a video is broadcast again. As a result, the waiting time is a very important factor. Access time can be reduced by repeatedly transmitting the whole movie through each of the given multiple server channels, and the multiple video streams are staggered evenly across the channels, similar to the **Staggered Broadcasting** scheme [42]. The advantage of this scheme is its simplicity and no local buffer space is needed at client sides. However, its access time will be reduced only in a linear fashion with increased bandwidth.

Other approaches to reductions in waiting time are based on partitioning a video into segments. In these approaches, clients need to buffer some segments of video in their set-top box (STB) while watching other segments. These broadcasting protocols are subdivided into three groups according to [51]. The revolutionary **Pyramid Broadcasting** (PB) protocol [43, 44], followed by the **Permutation-based Pyramid Broadcasting** (PPB) scheme [45] and the **Skyscraper Broadcasting** (SB) scheme [46], belongs to the first group. Protocols in this group of schemes partition each video into segments of increasing size and broadcast each segment during predefined periods on separate channels with equal bandwidth. The periodic broadcast of the first smallest segment is the most frequent allowing new requests to begin playback quickly. The periodic broadcast of each larger segment is scheduled on a different channel in a manner such that a client can always begin receiving the next larger segment during or immediately following the broadcast of a given previous segment. Clients need to download from at most two channels simultaneously and buffer a segment that is received earlier than needed for playback.

The segment sizes of a video in PB follows a geometrical series [d, αd, $\alpha^2 d$, $\alpha^3 d$, ...], where d is the size of the first segment, and α is equal to the ratio of the channel bandwidth to the video consumption rate ($\alpha > 1$). The drawback of PB is that each video segment requires a very high transmission rate, consequently client I/O bandwidth is also very high and client side buffers are usually more than 70% of the video program length.

PPB is proposed for addressing the client side issues in PB. PPB is similar to PB except that PPB further divides each segment into several blocks and multiplexes each segment channel into the same number of subchannels using a time division multiplexing method. The subchannels of each segment channel are staggered with each other to meet the same timing requirement as in PB.

If PB, PPB, and SB are compared, SB is the most efficient scheme. In SB, the segment size progression denoted as [d, 2d, 2d, 5d, 5d, 12d, 12d, 25d, 25d, 52d, 52d, ...] (d is the size of the first segment), offers the lowest latency and the client buffering space needed is only 20% of that needed by PPB. Moreover, SB employs low-bandwidth channels, each of which is at the playback rate.

Another scheme in the first group called the **Fast Broadcasting** (FB) protocol [47] is even more efficient than PB, PPB, and SB. In FB, a video is divided into geometrically increasing segment sizes of [d, 2d, 4d, ... $2^{K-2}d$, $2^{K-1}d$], where d is the size of the first segment. K is the total number of channels (segments), and the channel bandwidth is equal to the playback rate, the same as SB. Unlike PB, PPB, and SB, where clients need to download from at most 2 channels simultaneously, FB makes clients download from all K channels. FB incurs a $D/(2^K-1)$ waiting time, where D is the length of a whole video. The extra benefit of FB is that it provides heterogeneous users' service in terms of clients' buffer sizes; the larger the buffer size, the shorter the waiting time.

The most efficient scheme in the first group is the **Greedy Equal Bandwidth Broadcasting** (GEBB) protocol [48] which operates in a "greedy" fashion. GEBB receives as much of the data as possible from all of the channels immediately after tune-in and ceases receiving a segment immediately before playing it. The main difference between GEBB and all the above schemes in this group is that a fixed-delay policy is used in GEBB in which all clients need to wait for a small fixed delay before watching the selected video. The waiting time in GEBB is used to preload the first segment and simultaneously downloads part of the other segments rather than just waiting for the starting point of the first segment as in other schemes. Given the length of a video and a server bandwidth, the waiting time in GEBB approaches $D/(e^{K}-1)$ as the number of channels approaches infinity, where K is the ratio of the server bandwidth to the playback rate. GEBB is described in more detail in Chapter 2.

The second group of schemes is characterized by dividing the data for each video into equal-sized segments, repetitively transmitting them in separate channels of decreasing bandwidth, and allowing clients to be able to download from all the channels simultaneously. The *Harmonic Broadcasting* (HB) [49] protocol broadcasts each segment on a dedicated channel with bandwidth b/i, where b is the playback rate and i is the segment number (e.g., i=1 for the first segment). The bandwidth assignments for each successive segment follow the harmonic series: b, $b/2$, $b/3$, $b/4$, ... In HB a client must wait for the beginning of an instance of the first segment before the client can start receiving (and viewing) a video. Once the client starts receiving the first segment, the client will also start receiving all other channels dedicated to the video. HB can significantly reduce clients' waiting time and is proved to be optimal with respect to clients' waiting time given a specific transmission bandwidth [52].

Unfortunately, HB does not always deliver all data on time [53]. Paris et al. proposed the **Cautious Harmonic Broadcasting** (CHB), **Quasi-Harmonic Broadcasting** (QHB) [53] and **Poly-Harmonic Broadcasting** (PHB) [54] protocols to solve this problem. PHB and QHB are the most efficient in this group of schemes in terms of clients' maximum waiting time and average waiting time, respectively, given the total server bandwidth. Like GEBB, PHB uses "greedy" downloading and the fixed-delay policy. We'll introduce the detail in the next chapter too. The only drawback in this group of schemes is that the number of channels tends to infinity for getting a good latency and to handle so many channels with decreasing bandwidth is likely to be a daunting task.

To solve the problem of the HB-based schemes, Paris et al. further proposed the **Pagoda broadcasting** (PB) [55], **New Pagoda broadcasting** (NPB) [56] and **Fixed-delay Pagoda broadcasting** (FDPB) [57] protocols. They are the third group of schemes, which partition each video into a large number of small segments with equal size and uses time division multiplexing to periodically multiplex the segments into a small number of channels with equal bandwidth equal to the playback rate. To ensure that each segment is broadcast at the appropriate bandwidth, these schemes broadcast later segments less frequently instead of lowering channel bandwidth as HB-based schemes do.

In 2002, Tseng et al. and Bar-Noy et al. independently published the **Recursive Frequency-Splitting** (RFS) protocol [59] and a greedy algorithm for the harmonic windows scheduling problem [58] which packs as many segments as possible into a given number of channels with playback rate in a greedy way. To further reduce waiting time, Bar-Noy et al. introduced a shifting technique [60] for their greedy algorithm similar to FDPB, and their algorithm makes clients wait for a fixed waiting time before playing videos. We'll review the details of FDPB, RFS and Bar-Noy's greedy algorithm in the next chapter. Given the length of a video D and a server bandwidth, the three schemes, GEBB, PHB and FDPB, which all implement the fixed-delay policy, have the same lower bound for the maximum waiting time: $D/(e^{K}-1)$, where K is the ratio of the server bandwidth to the playback rate. None of the recently known broadcasting schemes can achieve maximum waiting times shorter than this lower bound, which we call ***the fixed-delay lower bound***.

To handle the case in which the clients' bandwidth is different from that of the server, Paris [57] improved his FDPB to adapt this case at the expense of clients' waiting time. A server divides a video into a different number of segments according to the clients' bandwidth limitation so that clients can simultaneously receive a limited number of channels rather than from all the video channels. **Generalized Fibonacci Broadcasting** (GFB) [61] is also proposed to address this case. Just like GEBB, GFB divides each video into segments of increasing size and broadcasts each of them in separate channels of equal bandwidth and the waiting time is used to completely download the first segment (and some other partial segments). However, unlike GEBB, GFB requires clients to download only from a given number of channels simultaneously other than from all the channels of a video. GFB, at the expense of the number of server channels, is more efficient than FDPB with a client bandwidth restriction in terms of clients' waiting time given client and the server bandwidth.

So far, all the above schemes are for a homogenous environment where all clients have the same bandwidth. The **HEterogeneous Receiver-Oriented** (HeRO) broadcasting protocol [62] employs one schedule for all clients with different bandwidths, and requires each client to wait until an appropriate time slot (every time slot corresponds to a broadcasting duration of the first segment) to start the video. The client bandwidth requirement at each time slot is different, so clients can download videos according to their bandwidth capacity at the possible expense of waiting time. HeRO belongs to the first group of schemes mentioned above.

1.2.4 Combination Scheme

In recent years, different combinations of services are also suggested to achieve cost-performance tradeoffs. Lee combines unicast and broadcast services in [63], in which multicast channels multicast hot movies periodically using the staggered broadcasting scheme and unicast channels provide patching channels for hot movies requests to reduce waiting time. Therefore, clients need to receive up to two video channels simultaneously, and require additional storage to cache part of the video. Storage capacity up to the length of the maximum waiting time of the corresponding staggered broadcasting schedule in multicast channels is required. Poon et al. [64] combines broadcast, multicast and unicast services together, and a video is delivered to customers through one of three kinds of channels, broadcast, multicast or unicast, depending on whether the video is very hot, hot or cold, respectively. In [64], efficient batching techniques for very hot and hot videos are studied while requests for old videos are served by dedicated channels and some VCR functions are also considered.

1.3 Contributions

In this thesis, we study broadcasting approaches and focus on the third group of schemes mentioned above. These schemes are normally used in a local area (distribution) network (LAN) to broadcast the top 10 or 20 hot movies. Since it is too time-consuming to find optimal solutions, we use heuristic algorithms to get near-optimal solutions in this thesis.

First, we propose a new broadcasting scheme, called **Generalized Fixed-Delay Pagoda Broadcasting** (GFDPB), which is based on the fixed-delay policy and improves Bar-Noy et al.'s greedy algorithm for the Harmonic Windows Scheduling Problem [58] as well as RFS [59] scheme. In GFDPB, clients need to wait for a small fixed delay to preload some segments from all channels simultaneously before watching the movie they have selected. Our GFDPB achieves the lowest maximum waiting time of all protocols using segments of equal duration and channels of equal bandwidth and of all currently known protocols given the same server bandwidth and the same number of channels. Furthermore, we present **Enhanced GFDPB** (EGFDPB) to perfect the overall performance of GFDPB when the number of channels is less than 4. Our result shows that when the number of channels is larger than 3, the results of GFDPB and EGFDPB are almost the same. Also, we analyze and propose an efficient server multiplexing and client demultiplexing scheme, and give an algorithm to translate the RFS representation for channel schedules used in [59] to the tree representation used in [58].

Second, we define the ***Harmonic Group Windows Scheduling* (*HGWS*)** problem and then present a new broadcasting scheme, the **Harmonic Page-set Broadcasting** (HPB) scheme, to solve it. The main idea of HGWS comes from my senior supervisor Dr. Kameda and his postdoctoral fellow Dr. Yi Sun. Our contributions to HGWS are that we use subchannels, a page-set schedule, and a page schedule to uniquely describe the problem and work out the slot-level details. A page-set in HPB corresponds to a segment in RFS or GFDPB (m=1) and is further divided into many consecutive pages. A page is the basic unit of the transmission of a video in HPB and all pages of a video have equal sizes and all pages in the same page-set have the same broadcasting period. The output of our algorithm is a page list, which is ready to be put into our server multiplexing scheme proposed in Section 3.7 for a real broadcasting. Our result shows that HPB provides the lowest average waiting time of all currently known protocols by using the least number of channels given the same server bandwidth. However, a drawback of HPB is that its maximum waiting time is twice its average waiting time, and much longer than that of GFDPB or FDPB.

Finally, to address HPB's drawback, we present our third new broadcasting scheme, a hybrid broadcasting scheme called the **Preloading Page-Set Broadcasting** (PPSB) scheme, which compromises the average waiting time and the maximum waiting time of HPB. While PPSB provides shorter average waiting time than the fixed-delay lower bound, and has the shortest average waiting time of all published broadcasting protocols in the third group, PPSB achieves much shorter maximum waiting time than HPB. From the probability point of view, PPSB provides a very desirable trade-off between the average waiting time and the maximum waiting time of all the published broadcasting protocols for a given server bandwidth, while guaranteeing that the maximum waiting time is not more than 32% longer than the fixed-delay lower bound. HPB's maximum waiting time is nearly 90% longer than the fixed-delay lower bound, and its average waiting time is just 9.5% shorter than that of PPSB. Furthermore, there are no currently published broadcasting protocols in the third group which can guarantee their average waiting time shorter than the fixed-delay lower bound for any given number of channels with equal bandwidth equal to playback rate.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we first review two most efficient broadcasting protocols in the first and second groups of broadcasting protocols for the purpose of comparison; then, we review some related schemes in the third group of broadcasting protocols. In Chapter 3, we first analyze and compare the RFS representation and the Tree representation of channel schedules; then, we propose two algorithms for GFDPB and its server multiplexing and client demultiplexing scheme, and we give some analysis and simulation results on the performance of GFDPB to show that it outperforms all the other broadcasting protocols. In Chapter 4, we first define the HGWS problem and present the HPB protocol to solve the problem; then, we propose PPSB, a hybrid broadcasting scheme, and analyze its performance. Finally, we conclude this thesis and discuss the future work in Chapter 5.

Chapter Two
Review

In Chapter 1, we have introduced three groups of broadcasting protocols. In this Chapter, to help understand the essence of the segment-scheduling problem, we review several broadcasting schemes in the third group. For the purpose of comparison, we also introduce GEBB, PHB and QHB, which are the most efficient broadcasting schemes in the first and second groups.

2.1 Basic Notation

Broadcasting protocols normally divide each video into a series of segments and transmit each segment periodically on dedicated server channels. While a client is playing a current video segment, it is guaranteed that the next segment is downloaded on time and the whole video can be played out continuously. In the third group of schemes, each video is divided into many fixed-size segments and all segments are periodically multiplexed in several channels with equal bandwidth equal to playback rate. In addition, each channel is partitioned into time slots of equal duration and the duration of each time slot is equal to the duration of each fixed-size segment.

For the convenience of expressing ideas and calculating formulas, we will use the following notations in our discussion:

K:	the number of broadcasting channels for each video
n:	the number of segments that each video is divided into
b:	the playback rate of a video in Mbps

b_i:	the i-th channel bandwidth

B:	the total bandwidth for each video in Mbps

B^*:	the normalized total bandwidth equal to B/b

w:	the maximum waiting time that clients may wait before watching the video

w^*:	the normalized waiting time equal to w/D

w':	the average waiting time

S_i:	the i-th segment (the first segment is S_1)

C_i:	the i-th channel (the first channel is C_0)

D:	the total length of a video in seconds

D_i:	the length of the i-th segment in seconds during playback in unequal-sized segmentation protocols

d:	the length of one segment in seconds during playback in equal size segmentation protocols; also, the duration of each time slot in the third group of schemes

m:	an integer $m \geq 1$. In fixed-delay protocols such as PHB, FDPB and our GFDPB, $w=md$; or in QHB, each time slot is divided into m subslots.

2.2 Greedy Equal Bandwidth Broadcasting (GEBB)

GEBB [48] is the most efficient protocol in the first group of broadcasting protocols which divide each video into segments of increasing size and transmit each of them in separate channels of the same bandwidth. GEBB operates in a "greedy" fashion, i.e. receives as much of the data as possible from all of the channels immediately after tune-in and finishes receiving a segment immediately before playing the corresponding segment. As shown in Figure 2.1, a client tunes in at t_0 and starts receiving immediately from all channels. After a fixed waiting time w, the client finishes receiving S_1 and immediately starts playing it. After playing S_1, the client finishes receiving S_2 and immediately starts playing S_2, and so on.

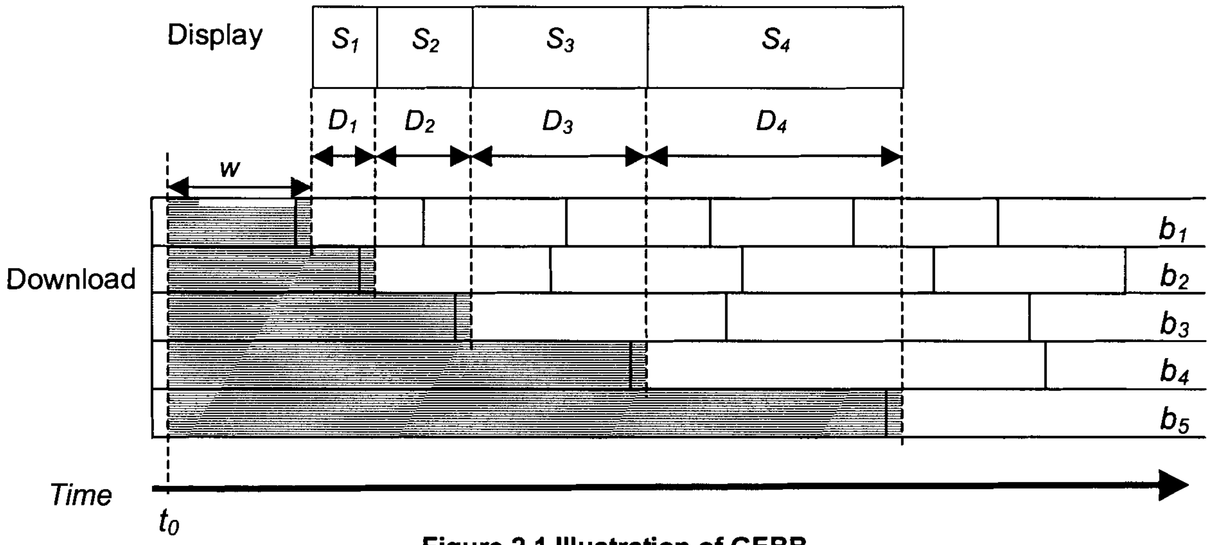


Figure 2.1 Illustration of GEBB

[48] describes an optimization problem as follows. Given a video with length D, the number of segments n (n=K), and the fixed waiting time w, the problem is to derive the segment durations and their corresponding channel bandwidths with the objective of minimizing the total server bandwidth required to broadcast a specific video. Formally, the problem can be stated as follows:

$$\text{minimize} \sum_{i=1}^{n} b_i \tag{2.2.1}$$

subject to

$$b_i\left(w+\sum_{j=1}^{i-1} D_j\right)=bD_i \qquad i=1,2,\ldots,n \tag{2.2.2}$$

$$\sum_{i=1}^{n} D_i = D, \qquad D_i > 0, \qquad b_i > 0 \qquad i = 1,2,\ldots, n$$

The condition represented by Formula (2.2.2) ensures that S_i is completely received at the exact time point when the play of S_{i-1} terminates. Thus, the segments of the video are available always and exactly on time for their playing.

From Formula (2.2.2), we can get a formula for *(1+b*$_i$*/b)*, and then multiplying all *n* of the *(1+*b_i*/b)* quantities produces

$$\prod_{i=1}^{n}(1+\frac{b_i}{b}) = \frac{w+\sum_{j=1}^{n} D_j}{w} = 1+\frac{D}{w} \qquad (2.2.3)$$

Thus, we have determined that the product of the n terms *(1+*b_i*/b)* is a constant since *D* and *w* are given. The minimization of $\sum_{i=1}^{n} b_i$ is equivalent to the minimization of $\sum_{i=1}^{n}(1+b_i/b)$ and it is well-known that the minimization of the sum of *n* terms given that their product is constant is achieved when all the *n* terms take the same value. Let us represent this optimal value as *(1+b*/b)*. Consequently, from Formula (2.2.3), we can derive that the channel bandwidth is

$$b_i = b^* = b(\sqrt[n]{\frac{D}{w}+1} - 1) \qquad i = 1,2,\ldots n \qquad (2.2.4)$$

Further from Formulae (2.2.2) and (2.2.4), we can derive

$$D_i = w\frac{b^*}{b}(1+\frac{b^*}{b})^{i-1} \qquad i = 1,2,\ldots, n \qquad (2.2.5)$$

Therefore, we can see that given *D, w,* and *n*, the segment size of GEBB follows a geometrical series. The total server bandwidth (*B*) necessary in GEBB for a particular video is *nb**.

$$B^* = B / b = n(\sqrt[n]{\frac{D}{w} + 1} - 1) \qquad (2.2.6)$$

It is straightforward to show that:

$$\lim_{n \to \infty} nb(\sqrt[n]{\frac{D}{w} + 1} - 1) = b \ln(\frac{D}{w} + 1) \qquad (2.2.7)$$

From Formula (2.2.6), we can get the formula for waiting time:

$$w = \frac{D}{(\frac{B^*}{n} + 1)^n - 1} \qquad (2.2.8)$$

Letting n increase to infinity, we can get the minimum waiting time for a given server bandwidth and a given number of channels:

$$w_{\min} = \lim_{n \to \infty} \frac{D}{(\frac{B^*}{n} + 1)^n - 1} = \frac{D}{e^{B^*} - 1} \qquad (2.2.9)$$

In the following sections, we will see that the waiting time of PHB as well as FDPB also asymptotically approaches this lower bound. It is shown in [48] that GEBB uses a smaller number of segments as well as a smaller number of channels than PHB to achieve the same fixed waiting time.

2.3 Poly-Harmonic Broadcasting (PHB)

PHB [54] is the most efficient protocol in terms of clients' maximum waiting time in the second group of broadcasting protocols which divide the data for each video into equal-sized segments and repetitively transmit them in separate channels of decreasing bandwidth. PHB, just like GEBB, uses greedy downloading. However, unlike GEBB, which uses increasing-sized segments during playback and equal channel bandwidth, PHB uses equal-sized segments during playback (display) and decreasing channel bandwidth. If we adapt Figure 2.1, by letting $D_1=D_2=\ldots=D_n$ and $b_1> b_2> b_3>\ldots> b_n$, the adapted Figure 2.1 becomes the illustration of PHB.

PHB breaks a video into n segments of (display) duration $d=D/n$ and separately broadcasts them in n channels. Under PHB, no client can start consuming the first segment of the video before having downloaded from all n channels during a time interval of duration $w=md$, where m is an integer $m \geq 1$. The waiting time is $w=mD/n$. As a result, segment S_i will not be consumed until $(m+i-1)d$ seconds have elapsed from the moment the client started downloading data from the server. Ensuring that segment S_i will be entirely broadcast over this time interval suffices to guarantee that all the content of segment S_i will be already loaded in the set-top box before the client starts viewing that segment. This can be achieved by retransmitting segment S_i at a transmission rate $b_i=b/(m+i-1)$, since each segment has an equal size bd. Therefore, the total bandwidth required by PHB is given by

$$B_{PHB}(n,m)=\sum_{i=1}^{n} b_i = b\sum_{i=1}^{n}\frac{1}{m+i-1}=b(H(n+m-1)-H(m-1)) \qquad (2.3.1)$$

where $H(i)$ represents the i-th Harmonic number.

If $k=n/m$ and k is an integer, where $k\geq 1$, Formula (2.3.1) can be rewritten as

$$B_{PHB}(k,m)=b(H((k+1)m-1)-H(m-1))$$

[54] derives $B_{PHB}(k,m+1)<B_{PHB}(k,m)$ for all $k\geq 1$ and $m\geq 1$. This means increasing m and n while keeping k constant will always result in a reduction in total

bandwidth. Thus, when m and n go to infinity while k remains constant, we can compute the limit of $B_{PHB}(K,m)$ and derive a lower bound for the total bandwidth required by PHB.

$$\lim_{n,m \to \infty,\ n/m = k} B_{PHB}(k,m) = \lim_{n,m \to \infty,\ n/m = k} \sum_{i=1}^{n} \frac{b}{m+i-1} = \int_{0}^{D} \frac{b}{w+t} dt = b \log\left(\frac{D}{w}+1\right) \qquad (2.3.2)$$

We can see Formula (2.3.2) is exactly the same as the GEBB lower bound Formula (2.2.7). However, GEBB is more efficient than PHB in terms of the number of channels required for a fixed waiting time.

2.4 Quasi-Harmonic Broadcasting (QHB)

QHB [53] is the most efficient protocol in terms of clients' average waiting time in the second group of broadcasting protocols [65]. Like other harmonic protocols, QHB divides each video into n equal-sized segments and repetitively transmit them in separate channels of decreasing bandwidth. The first segment is broadcast repeatedly on the first channel with the display rate b. However, each segment i, $1< i \leq n$, is broken into $im-1$ fragments for some parameter m, and a client will receive m fragments from each channel per time slot. If we divide each time slot into m equally sized subslots, the client will receive a single fragment during each subslot.

In each channel except the first, the last subslot of each time slot is used to transmit the first $i-1$ fragments of S_i in order. The rest of the subslots transmit the other $i(m-1)$ fragments such that the k^{th} subslot of the j^{th} slot is used to transmit fragment $ik+((j-1) \bmod i)$ of S_i. For example, in Figure 2.2, the third segment is subdivided into eight fragments that occupy three slots, each comprising three subslots. The second subslot of the fourth slot of the third channel, which broadcasts S_3, is used to transmit fragment 3x2+ ((4–1) mod 3) =6 of S_3, i.e., fragment $S_{3,6}$. It is proved in [53] that QHB delivers all video data on time.

1st channel	S_1			S_1			S_1			S_1		
2nd channel	$S_{2.2}$	$S_{2.4}$	$S_{2.1}$	$S_{2.3}$	$S_{2.5}$	$S_{2.1}$	$S_{2.2}$	$S_{2.4}$	$S_{2.1}$	$S_{2.3}$	$S_{2.5}$	$S_{2.1}$
3rd Channel	$S_{3.3}$	$S_{3.6}$	$S_{3.1}$	$S_{3.4}$	$S_{3.7}$	$S_{3.2}$	$S_{3.5}$	$S_{3.8}$	$S_{3.1}$	$S_{3.3}$	$S_{3.6}$	$S_{3.2}$
Slot	1st			2nd			3rd			4th		
Subslot	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd

Figure 2.2 QHB schedule for *K*=3, *m*=3

We can see that each segment S_i, $i>1$, broadcasts in $(i-1+\frac{m-1}{m})$ time slots. Because each segment has an equal size and the first channel bandwidth is *b,* we can easily derive the bandwidth for each channel.

$$\begin{aligned} b_i &= b & & if \quad i = 1 \\ b_i &= \frac{bm}{im - 1} & & otherwise \end{aligned} \tag{2.4.1}$$

Then the total bandwidth B_{QHB} required by QHB is given by

$$B_{QHB} = b + \sum_{i=2}^{n} \frac{bm}{im-1} = bH(n) + \sum_{i=2}^{n} \frac{b}{i(im-1)} \tag{2.4.2}$$

The maximum waiting time is *w=D/n* and the normalized average waiting time is 1/2*n*.

2.5 Fixed-Delay Pagoda Broadcasting (FDPB)

FDPB [57] and RFS, which will be introduced in Section 2.6, belong to the third group of broadcasting protocols, which are pagoda-based protocols. They partition each video into *n* equal-sized segments of duration *d=D/n,* use time division multiplexing to broadcast these segments at different frequencies over *K* channels with equal bandwidth equal to the video consumption rate *b*, and let each segment transmission occupy a time *slot* of duration *d*. In addition, clients need to simultaneously download from all *K* channels.

Like GEBB and PHB, the FDPB protocol implements the fixed-delay policy and requires all clients to wait for a fixed time interval $w=md$, where m is an integer $m \geq 1$. Thus, segment S_1 needs to be transmitted at least once every m slots and must always be received before a customer, after *md*-second wait and preload, starts watching the video. More generally, segment S_i needs to be transmitted at least once every $m+i-1$ slots to guarantee that S_i has been buffered before it is needed to be consumed.

The FDPB protocol partitions each channel C_i into s_i subchannels in such a way that slot j (slot number starts from 0) of channel C_i belongs to its subchannel (j mod s_i). Thus, each subchannel has $1/s_i$ of the slots and $1/s_i$ of the bandwidth of Channel C_i. Figure 2.3 shows how a channel is partitioned into three subchannels.

FDPB maps segments into subchannels in a strictly sequential fashion. The first segments of a video are mapped into subchannel 0 of channel 1, the next segments into subchannel 1 of the same channel, and so on until all s_1 subchannels have been used. The process repeats itself for the subchannels of channel C_2 to C_K.

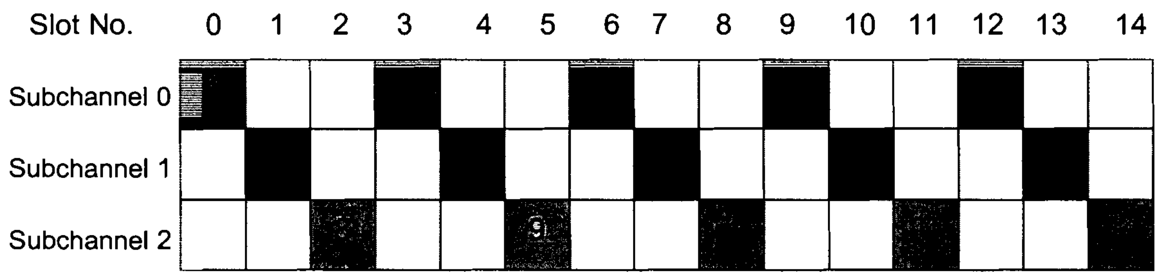


Figure 2.3 A channel partitioned into 3 subchannels in FDPB

By trial and error, Paris [57] found the optimal mapping of subchanels for a given channel C_i is always achieved when channel C_i is partitioned into $\sqrt{m+j-1}$

subchannels assuming that the first segment assigned to this channel is segment S_j. Consider, for instance, the case when m=9. As figure 2.3 indicates, channel C_0 will be partitioned into $\sqrt{9+1-1}=3$ subchannels since the first segment assigned to C_0 is S_1. Since m=9, segment S_1 needs to be repeated at least once every nine slots and we assign S_1 to subchannel 0. Since subchannel 0 occupies 1/3 of the slots of channel C_0, we can map up to 9/3=3 segments into it, while ensuring that each of these three segments (S_1 - S_3) will be repeated once every nine slots (see Figure 2.3). The first segment assigned to subchannel 1 is S_4, which needs to be repeated at least once every 9+4–1=12 slots. As a result, we can map 12/3=4 segments into suchannel 1 while ensuring that each of these four segments (S_4 - S_7) will be repeated once every 12 slots. The first segment assigned to subchannel 2 is S_8, which needs to be repeated at least once every 9+8–1=16 slots. Thus, we can map $\left\lfloor \frac{16}{3} \right\rfloor = 5$ segments into subchannel 2 while ensuring that each of these five segments (S_8-S_{12}) will be repeated once every 5*3=15 slots. As a result, channel C_0 will transmit a total of 12 segments. We can repeat this procedure for channel C_1 where the first segment assigned is S_{13}, so the number of subchannels is rounded off to $round(\sqrt{9+13-1})=5$, and so on.

After getting the number of segments n which are mapped into the given K channels, we can calculate the waiting time for FDPB by formula $w=mD/n$, which is exactly the same as that for PHB. PHB broadcasts all segments in separate channels of decreasing bandwidth, whereas FDPB broadcasts them in decreasing frequencies over K channels with equal bandwidth equal to the video consumption rate.

Recently, [65] points out that $\sqrt{m+i-1}$ is not always the optimal number of subchannels for a channel C_i, where the first segment assigned is S_i. They find that the actual optima are laid between $\max\{\sqrt{m+i-1}-5,\ 1\}$ and $\sqrt{2.5(m+i-1)}+5$. However, the improvement is small and at most around 1%.

[57] also derives a lower bound, which is the same as Formulae (2.3.2) and (2.2.7), for the total bandwidth required by those schemes implementing the fixed-delay policy as follows:

$$B_{\min} = b \log \frac{D + w}{w} \tag{2.5.1}$$

From Formula (2.5.1), [57] derives a lower bound for the maximum waiting time when the broadcasting bandwidth is equal to K times the video consumption rate,

$$w_{\min} = \frac{D}{e^{K} - 1} \tag{2.5.2}$$

We call this lower bound the ***fixed-delay lower bound***. No currently known broadcasting protocols can achieve maximum waiting time lower than this lower bound.

2.6 Recursive Frequency-Splitting (RFS) Scheme

As mentioned in Section 2.5, RFS [59], like FDPB, belongs to the third group of broadcasting protocols. However, unlike GEBB, PHB and FDPB, RFS does not implement the fixed-delay policy. In RFS, channel C_0 continuously repeats segment S_1 to ensure that it is repeated in every slot, so a client should wait until the beginning of any new time slot to start watching a video and at the same time to start the buffering process while downloading simultaneously from all K channels. Thus, its maximum waiting time is $w=D/n$.

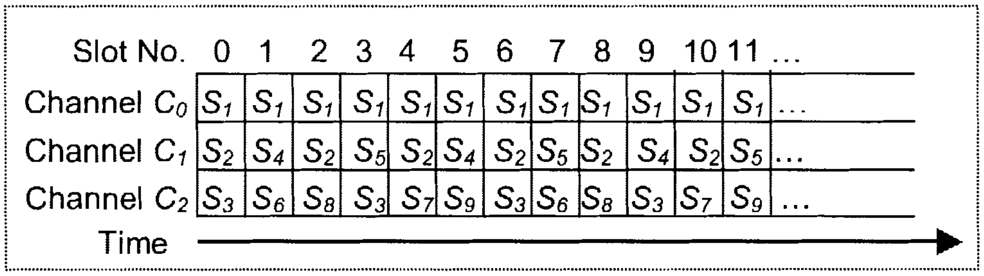
In RFS, generally, segment S_i must be broadcast at least once on one of K channels in every consecutive i time slots to guarantee that S_i has been received or will be received at the time slot when a viewer needs to consume it. Thus, segment S_i must be broadcast on one of K channels periodically with a frequency no less than $1/i$. [59] introduces the concept of "periodical time slots" as follows: a ***slot sequence*** $SS(C_i, s, p)$ is an infinite sequence of time slots $[s, s+p, s+2p, \ldots]$ belonging to channel C_i, beginning at slot s, which we will call the ***start slot*** of the slot sequence or of the assigned segment, and repeating infinitely with a period of p slots, where C_i is one of the K channels, $p \geq 1$ is an integer, and s is an integer satisfying $0 \leq s \leq p-1$. Thus, when $p=1$, the time slots of the corresponding slot sequence will be continuous and the slot sequence represents a complete channel (e.g., $C_i = SS(C_i, 0, 1)$).

```
Input: a set of $K$ channels $C_0, C_1, \ldots, C_{K-1}$
Output: $n$, and the assignment of one slot sequence for each segment $S_i$, $i=1,\ldots n$.
1) Let POOL be a set of free slot sequences:
   POOL = $\{SS(C_0, 0, 1), SS(C_1, 0, 1), \ldots, SS(C_{K-1}, 0, 1)\}$
   Intuitively, this is the set of free channels $C_0, C_1, \ldots, C_{K-1}$ that are given initially.
2) Initialize $n=1$ for the first segment.
3) Pick a slot sequence $SS(C_i, s, p) \in$ POOL, such that $p \leq n$. If more than one
   sequence in POOL satisfies this condition, choose the sequence(s) with the
   smallest ($n$ mod $p$) and break a tie by selecting the largest $p$, and then do the
   subtraction POOL=POOL $- \{SS(C_i, s, p)\}$.
4) Split $SS(C_i, s, p)$ into $\alpha = \lfloor n/p \rfloor$ slot sequences: $SS(C_i, s, \alpha p)$, $SS(C_i, s+p, \alpha p)$,
   $SS(C_i, s+2p, \alpha p), \ldots, SS(C_i, s+(\alpha-1)p, \alpha p)$. Assign $SS(C_i, s, \alpha p)$ to $S_n$; then, do the
   union POOL=POOL $\cup \{SS(C_i, s+jp, \alpha p), j=1,2,\ldots, \alpha-1\}$
5) If there exists a slot sequence $SS(C_i, s, p) \in$ POOL such that $p \leq n+1$, then
   increase $n$ by 1 and go to step 3 to schedule the next segment; otherwise,
   terminate this procedure and output the value of $n$.
```

Figure 2.4 RFS Algorithm

The RFS scheme is based on a concept called "frequency splitting". For segment S_j, we should allocate a slot sequence $SS(C_i, s, p)$ such that $p \leq j$. It is desirable that the value of p be as close to j as possible, since a larger p means less waste in communication bandwidth. The best case is $p = j$. However, when $p < j$ and $j/p \geq 2$, we need to partition $SS(C_i, s, p)$ into $\alpha = \lfloor j/p \rfloor$ subsequences with the same period of αp time slots, as indicated in step 4 of Figure 2.4, and assign one of these subsequences to S_j. The period αp is the maximal period that is a multiple of p and not larger than j. According to this concept of frequency splitting, [59] gives the RFS algorithm as shown in Figure 2.4.

Unlike FDPB which maps segments into subchannels and channels in a strictly sequential fashion, RFS maps segments in a "greedy" fashion in order to minimize the waste of bandwidth. In the procedure shown in Figure 2.4, authors of [59] try to increase the value of n repeatedly. Step 3 is a heuristic for reducing the waste of bandwidth when performing the assignment in step 4, and it is also a heuristic for leaving more flexibility in subsequent assignments. Step 4 performs the splitting. Step 5 checks whether the next segment can be accommodated. Figure 2.5 shows the result of running the RFS scheme for K = 3. We can express the result in a slot sequence expression: SS(C_0, 0, 1, S_1), SS(C_1, 0, 2, S_2), SS(C_1, 1, 4, S_4), SS(C_1, 3, 4, S_5), SS(C_2, 0, 3, S_3), SS(C_2, 1, 6, S_6), SS(C_2, 2, 6, S_8) SS(C_2, 4, 6, S_7), and SS(C_2, 5, 6, S_9), where the last parameter in each bracket is what we add to represent the segment assigned to this slot sequence.



Slot No.	0	1	2	3	4	5	6	7	8	9	10	11	...
Channel C_0	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	...
Channel C_1	S_2	S_4	S_2	S_5	S_2	S_4	S_2	S_5	S_2	S_4	S_2	S_5	...
Channel C_2	S_3	S_6	S_8	S_3	S_7	S_9	S_3	S_6	S_8	S_3	S_7	S_9	...
Time													

Figure 2.5 RFS scheme result for *K=3* channels

[59] gives an upper bound of the number of segments n that can be packed into K channels using RFS. Upper bound n must satisfy

$$\sum_{i=1}^{n} \frac{1}{f(i)} \leq K < \sum_{i=1}^{n+1} \frac{1}{f(i)} \tag{2.6.1}$$

$$where \quad f(i) = \begin{cases} i-1, & \text{if } i \text{ is a prime number and } i \geq p(K) \\ i, & \text{otherwise.} \end{cases}$$

and $p(i)$ is the *i*th prime number (i.e., *p*(1)=2, *p*(2)=3, *p*(3)=5, etc.). Thus, given a video with duration *D* seconds, *D*/*n* is a lower bound on the maximum waiting time.

We summarize the proof of the above upper bound as follows. We know segment S_i must be broadcast at least once in every continuous *i* time slot. According to Lemma 3 of [59], no two segments S_i and S_j, such that *i* and *j* are primes, can be broadcast in the same channel with period *i* and *j*, respectively. Under the best situation, we may place each of the *K* – 1 segments $S_{p(1)}$, $S_{p(2)}$, • • •, $S_{p(k-1)}$ in a separate channel by broadcasting them with perfect periods *p*(1), *p*(2), • • •, *p*(*K*-1), respectively. The other segments $S_{p(K)}$, $S_{p(K+1)}$, $S_{p(K+2)}$, • • •, each of which has a segment number equal to a prime and larger than *p*(*K*–1), will each be forced to broadcast with a smaller period less than its segment number no matter which channels they broadcast on (otherwise, conflict will occur). Therefore, in the best case, segment S_i has period *f*(*i*) in Formula (2.6.1) and will consume at least 1/*f*(*i*) of the channel bandwidth. Summing this over all segments gives Formula (2.6.1).

Chapter Three
Generalized Fixed-Delay Pagoda Broadcasting

In the previous chapter, we reviewed the RFS and FDPB schemes, which are the most bandwidth-efficient schemes that are currently known in the third group of broadcasting schemes, i.e., those which use equal-sized segments. In Section 3.1, we will introduce a tree representation of a channel schedule and a window scheduling algorithm, which is similar to RFS but is represented in terms of a tree. Because it is more intuitive, we will use the tree representation extensively throughout the rest of the thesis. In Sections 3.2 to 3.4, we will propose our two versions of GFDPB and analyze their performance. In Sections 3.5 and 3.6, we will present and analyze the efficient multiplexing and demultiplexing methods to be used by the server and clients, respectively. Finally, in Section 3.7, we will present an algorithm to translate the slot sequence representation for a channel schedule to the tree representation.

3.1 Previous Work

Bar-Noy et al. proposed a greedy algorithm for the ***Harmonic Windows Scheduling*** (HWS) problem [58] similar to RFS but it uses a different representation and a different point of view. A ***window*** consists of a set of consecutive time slots and the number of time slots in the window is called the ***window size***. HWS uses equal-sized segments called *pages* [58] such that a page exactly fits in a time slot. For conformance with other broadcasting schemes in the third group we use the term segment instead of page. The *optimal harmonic windows scheduling problem* is defined as follows:

Given K slotted channels, what is the maximum number of segments that can be scheduled on the K channels, one segment per channel at each time slot, such that segment i appears at least once in every window of size i?

In this thesis, a segment appears only repeatedly on a single channel. A schedule is said to be ***perfect*** if it schedules each segment at fixed slot intervals. In this case, the size of the interval is called the ***period*** of the segment.

A *channel schedule* is defined as a sequence of segments to be transmitted on a single channel such that the length of the sequence is equal to an integer multiple of the least common multiple (LCM) of the periods of all the segments broadcast in this channel. In Figure 2.5, the 2nd channel broadcasts S_2, S_4 and S_5, whose periods are 2, 4, and 4 time slots, respectively, so that the LCM is 4. Therefore, the shortest channel schedule for the 2nd channel is <S_2, S_4, S_2, S_5> or any of its three cyclic shifts i.e., <S_5, S_2, S_4, S_2>, <S_2, S_5, S_2, S_4>, and <S_4, S_2, S_5, S_2>. The 3rd channel broadcasts S_3, S_6, S_8, S_7, and S_9, whose periods are 3, 6, 6, 6, and 6 time slots, respectively, so that the LCM is 6. Therefore, the shortest channel schedule for the 3rd channel is <S_3, S_6, S_8, S_3, S_7, S_9> or any of its five cyclic shifts. Each channel repeats its channel schedule forever and any repetition of a channel schedule is also a channel schedule by definition.

Instead of using many slot sequences to represent a broadcasting schedule for each channel and each of the slot sequences to represent a segment broadcasting schedule as RFS does, Bar Noy et al. use a tree representation such as shown in Figure 3.1, in which each leaf determines a segment's start slot (offset) and broadcasting period in the channel. Therefore, each node in the tree representation, except the root, corresponds to a slot sequence in RFS, and the child nodes of each non-leaf node correspond to the subsequences resulting from the frequency splitting of such a node.

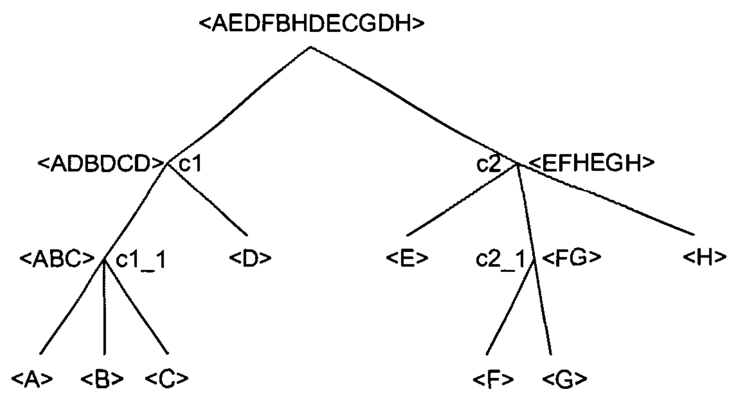


Figure 3.1 An example of a tree for segments A, B, C, D, E, F, G, H

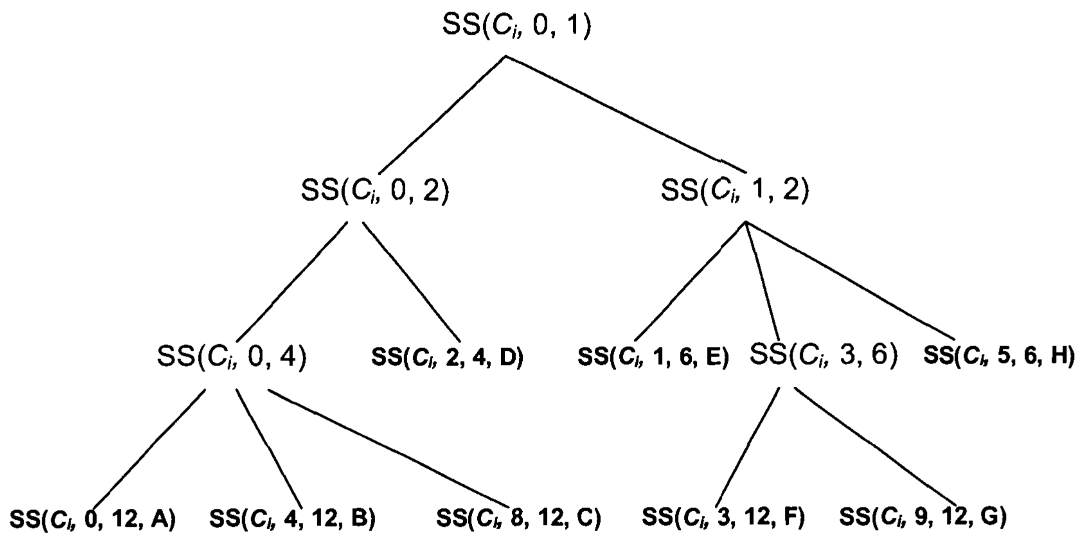


Figure 3.2 Frequency splitting representation corresponding to Figure 3.1

Figure 3.1 represents a certain channel schedule in the tree representation. Let's call the channel involved channel C_i, which is assigned to broadcast segments *A, B, C, D, E, F, G, H*. Now we can see the correspondence between a tree representation and the slot sequence representation. At the very beginning, this tree only has a root node corresponding to *SS*(C_i, 0, 1) mentioned in Section 2.6, and then this root splits into two child nodes c1 and c2, corresponding to SS(C_i, 0, 2) and SS(C_i, 1, 2), respectively, as shown in Figure 3.2. Node c1 splits again into two child nodes. One of its child nodes is c1_1, corresponding to SS(C_i, 0, 4), and the other represents segment *D*, corresponding to SS(C_i, 2, 4, D). Node c2 splits again into three child nodes. One of its child nodes represents segment E, corresponding to SS(C_i, 1, 6, E), and another represents segment H, corresponding to SS(C_i, 5, 6, H). SS( ) with four parameters represents a slot sequence allocated to a segment, and SS( ) with three parameters represents a non-leaf node of the tree, which will be split further. The other child node is c2_1, corresponding to SS(C_i, 3, 6). Node c1_1 splits again to three child leaves and node c2_1 splits again to two child leaves. The whole frequency splitting procedure is illustrated in Figure 3.2.

Procedure: Tree-to-Schedule

Input: A tree T with n leaves that are labeled with the labels $p_1, \ldots, p_n$ that represent the segments. Let T have $d \geq 0$ subtrees.

Output: A perfect channel schedule S' for these segments in which each segment x_j has a fixed period

Base case: If n =1 (it is a leaf tree), then the schedule is S'=< p_1>

The recursive step: If n_i >1 and therefore d > 0, do the following:

a) Recursively construct the channel schedules $S'_1, \ldots, S'_d$ of all the d subtrees of T. Assume their respective lengths are $l_1, \ldots, l_d$.

b) Replicate each channel schedule and make all of them have the same length l' = LCM{ $l_1, \ldots, l_d$}. Let the new schedules be $S^*_1, \ldots, S^*_d$.

c) The final channel schedule S' of length $l = dl'$ is constructed by alternately picking l' times the next segment from the d channel schedules $S^*_1, \ldots, S^*_d$.

Figure 3.3 Algorithm for translating a tree to channel schedule

[58] provides a recursive procedure for translating a tree to its channel schedule as shown in Figure 3.3. The *round-robin tree* is a simple example. This is a tree whose root has d children, all of which are leaves. If they are labeled by the segment numbers $p_1, p_2, \ldots, p_d$, then the channel schedule represented by this tree is the round-robin channel schedule S=<$p_1, p_2, \ldots, p_d$>. Since this channel schedule simply goes through the d children of the root in order, we call it round-robin. A more complicated example is the tree in Figure 3.1. Its root has two subtrees (d=2) and the labels of the leaves are A,

B, …, H. Applying procedure Tree-to-Schedule on the left and right subtrees yields the channel schedules *<ADBDCD>* and *<EFHEGH>*, respectively. Hence, the schedule represented by the tree is *<AEDFBHDECGDH>*. By repeating the above channel schedule infinitely, we can see the start slot and the period of each segment is exactly the same as depicted in Figure 3.2. For example, *H* starts from slot 5 and its period is 6 slots. Similarly, *G* starts from slot 9 and its period is 12 slots.

From the recursive round-robin character of the algorithm in Figure 3.3 and the correspondence between a tree representation and a slot sequence representation, we can easily find that the period of a segment (label) in a tree representation is equal to the product of the numbers of children of the segment's ancestors. For example, for the leaf labeled G in Figure 3.1, the numbers of children of all the ancestors of the leaf, from the root to its parent c2_1, are 2, 3, 2 in this order, so segment G's period is 2x3x2=12.

Now we introduce the greedy algorithm proposed in [58]. The authors of [58] first define two kinds of trees. One is an ***open tree*** which is a tree whose leaves are labeled with two types of labels: a segment label and a window label. The other is a ***closed tree*** which is a tree whose leaves have only segment labels. The leaves with segment labels have been assigned to specific segments and those with window labels are free leaves, or the leaves which have not been assigned to specific segments, and are labeled by their periods. The period of a window label or a leaf is calculated in the same way as that of a segment label mentioned above. An ***open forest*** is a collection of open trees and a ***closed forest*** is a collection of closed trees. Initially, all the *K* trees are singleton open trees with window labels whose value is 1. The greedy algorithm terminates when all the *K* trees become closed trees, which means no free slot sequence is available to be allocated to any more segment.

```
Input: A set of $K$ channels $C_0, C_1, \ldots, C_{K-1}$.
Output: $n$ and $K$ closed trees with a total of $n$ leaves labeled with segment labels,
    each of which represents segment $i$, $i=1,\ldots n$
1) Initialize $K$ channels into $K$ singleton open trees with window labels 1.
2) Initialize $r=1$, where $r$ indicates the $r$-th segment.
3) Let $w_1 \le w_2 \le \cdots \le w_k$ be the ordered list of the labels of all the leaves in the
    forest whose labels are of type window (initially they are all 1).
4) Let $m_j = (r \bmod w_j)$ for $1 \le j \le k$.
5) Let $i$ be the index such that $m_i$ is the minimum among all the $m_j$. Break a tie by
    selecting the index that is associated with the largest window label.
6) Let $d_i = \lfloor r / w_i \rfloor$ and $T_s$ be the tree that contains $w_i$.
7) If $d_i = 1$, then replace the window label $w_i$ with the segment label $r$ in the tree $T_s$.
    (This operation is called the replacement operation). Otherwise, add $d_i$ children to
    the leaf associated with $w_i$ replacing this leaf with $d_i$ new leaves. The first child is
    labeled with $r$ and the rest are labeled with the window label $w_i \cdot d_i$ which are put in
    order into the window label list. (This operation is called the split operation).
8) If the window label list is not empty, then increment $r$ and go to step 3 to
    schedule the next segment; otherwise, terminate this procedure and output the
    value of $n=r$ and the $K$ closed trees.
```

Figure 3.4 Greedy algorithm for Harmonic Windows Scheduling problem

[58] gives the greedy algorithm as shown in Figure 3.4, in which the *split* operation in step 7 is the same as the "frequency splitting" operation of RFS. Variables d_i and w_i here are respectively the same as α and p in Figure 2.4. In [58], the authors propose two possible modifications to the split operation and only one of them is *manipulable*. The manipulable modification tries to leave leaves open with small window labels as long as possible and works as follows. When d_i (in step 6 of Figure 3.4) is a composite number, the split operation is carried out in several steps in the increasing order of the prime factors of d_i. For example, let d_i = 12 for a node with a window label w_i. The prime factors of 12 are 2, 2, and 3; therefore, the node is split twice into two nodes in the first two split operations, and once into three nodes in the last split operation, as shown in Figure 3.5, where all labels are window labels. Thus, it creates five new leaves whose window labels are $2w_i$, $4w_i$, $12w_i$, $12w_i$, $12w_i$, respectively. The

leftmost window label $12w_i$ in Figure 3.5 produced in the last split operation becomes a segment label but the rest remain window labels. They found that the basic greedy algorithm with this modification does not always get a better result. In this thesis, we call the basic greedy algorithm with the modification, i.e., the multilevel split for composite d_i, the ***multilevel splitting greedy algorithm***.

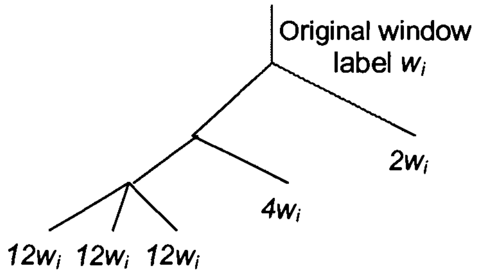


Figure 3.5 Multilevel splitting

We can translate each segment label in the tree representation to a (slot) sequence representation. First, we just copy the channel number and the segment label which is indicated by its segment number. Second, we must calculate the start slot of the segment, which is called the *offset* of the segment in [61] by Bar-Noy et al. Third, we must calculate the segment's actual period. In [61], Bar-Noy et al. give the formulae for the calculation of the start slot and the actual period. Both formulae are given below and they are proved in [61].

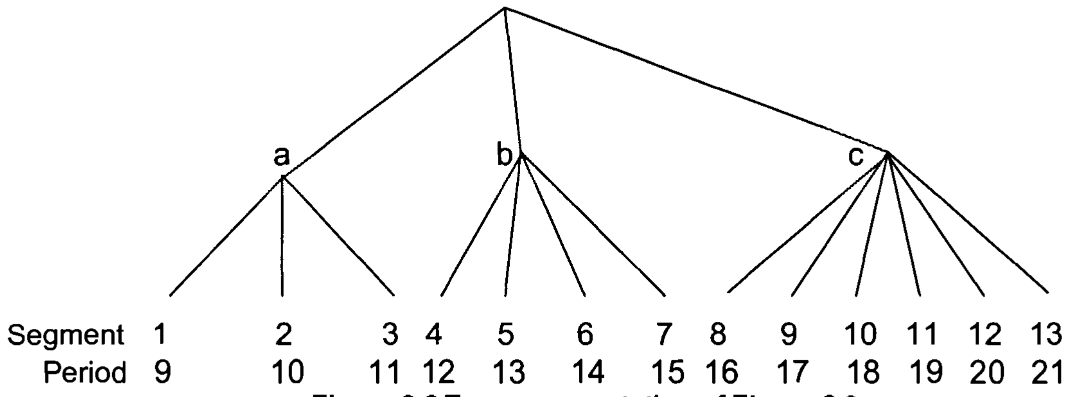
For each node J, let $d(J)$ denote the number of children of J, and let J' denote the parent of a non-root node J. They first define the period $T(J)$ of node J inductively as follows: for the root node $J0$, $T(J0) = 1$, and for a non-root node J

$$T(J) = T(J') \cdot d(J'). \qquad (3.1.1)$$

Thus, by definition, for a leaf J, its period $T(J)$ is the product of the number of children of each of its ancestors. Next they assign an ***offset*** (start slot) for each leaf. If a node has d children, they are numbered 0, 1, . . . , d - 1, left to right. Each node J is associated with a number $h(J)$, which is the number of its left siblings. The offset can be computed recursively as follows: $a(J0) = 0$ for the root node $J0$, and for each node J with parent J',

$$a(J) = a(J') + h(J)\ T(J') \tag{3.1.2}$$

Take segment G in Figure 3.1 for instance. Its period is 2x3x2=12 and its offset (start slot) is equal to a(c2_1) + 1x6 = (a(c2) + 1x2) + 1x6 = ((a($J0$) + 1x$T(J0)$) +1x2) + 1x6 = 9, where $J0$ is the root.


a
b
c
Segment 1 2 3 4 5 6 7 8 9 10 11 12 13
Period 9 10 11 12 13 14 15 16 17 18 19 20 21
Figure 3.6 Tree representation of Figure 2.3

In [60], Bar-Noy et al. propose the RR^2 algorithm which is almost exactly the tree version of FDPB. The main difference is that the optimal number of subchannels for the channel in which the first segment assigned is S_i is found by nearly exhaustive checks from 2 to $(m+i-1)/2$ instead of using the integer closest to $\sqrt{m+i-1}$ as in FDPB. RR^2 represents each channel schedule of FDPB as a tree of height 2, as shown in Figure 3.6 for the channel schedule in Figure 2.3. In Figure 3.6, the three internal nodes at level 1 represent the three subchannels and the leaves at level 2 represent the segments in each subchannel. The authors also adapt the greedy algorithm mentioned above to the fixed-delay policy in the simulation section of [60] for a small number of channels and a small number of m.

3.2 Generalized Fixed-Delay Pagoda Broadcasting

In this section, we present our GFDPB scheme which improves Bar-Noy et al.'s greedy algorithm. GFDPB, like FDPB, is based on the fixed-delay policy. In GFDPB, a video of duration D is broadcast over K channels $\{C_i \mid 0 \leq i < K\}$, each with bandwidth equal to the video consumption rate b. Each video is partitioned into n equal-sized segments of duration $d = D/n$. These n segments are broadcast at different frequencies over the K channels, and each segment transmission occupies a *slot* of duration d. A client in GFDPB needs to wait for a fixed time interval $w=md$ before starting to display a video, where m is some integer $m \geq 1$. As in FDPB, segment S_i in GFDPB needs to be transmitted at least once every $m+i-1$ slots to guarantee that S_i has been buffered before it is needed.

Maximizing the number of segments scheduled in the given K channels, such that segment i appears at least once in every window of size $m + i - 1$, is called the **optimal truncated-Harmonic scheduling problem** in [65].

We call $m+i-1$ the ***ideal period*** of segment S_i. The ideal period *idealP* of a segment S_i of a video is an integer such that segment S_i needs to be repeated at least once every *idealP* slots to ensure the continuity of the display of the video. In RFS, *idealP*=i; whereas in FDPB, *idealP*=$m+i-1$. The actual period of a segment in each broadcasting schedule must be equal to or less than its ideal period.

However, unlike FDPB which maps segments into subchannels and channels in a strictly sequential fashion, GFDPB, like RFS and the greedy algorithm, maps segments in a "greedy" fashion in order to minimize the waste of bandwidth and to pack more segments into a given number of channels.

If we change the greedy algorithm to conform to the fixed-delay policy directly, we need to replace "Initialize r=1" in step 2 of Figure 3.4 with "initialize r=m", add m to the input, and replace "output the value of n=r" in step 8 with "output the value of n=r–m+1". In the same way, we can adapt the RFS algorithm to the fixed-delay policy. By careful observation of results of the greedy algorithm under the fixed-delay policy, we found that if $d_i = \lfloor r / w_i \rfloor$ in step 6 of Figure 3.4 is a big integer, then the free leaf with the window label w_i will be split into a large number of leaves with the same large window label $w_i \bullet d_i$. This leads to inflexibility in processing the subsequent segments after the split. For example, for K=1 and m=100, in processing segment S_1, channel C_0 is split into 100 leaves with the same period 100, so the maximum number of segments that can be packed into this channel is, according to the basic greedy algorithm, only 100.

For a leaf with a window label w_i, the replacement operation or the split operation of the leaf will not cause bandwidth loss only if it is assigned to a segment S_i whose ideal period m+i-1 is exactly equal to w_i or an integer multiple of w_i,; otherwise, the actual period of segment S_i, or $w_i \bullet d_i$, should be less than the segment's ideal period since $d_i = \lfloor (m+i-1)/w_i \rfloor < (m+i-1)/w_i$. If we allocate too much bandwidth to S_i, i.e., $d_i \bullet w_i < m+i-1$, we ***lose bandwidth at segment S_i*** or at the window label w_i.

For a window label w_i, we don't know in advance which segment label will replace it or split it. If the probability of replacing or splitting is the same for all segments, then the probability to lose bandwidth at this window label is (1-1/w_i). For any segment r, the possible remainders m_i =(r mod w_i) are 0, 1, ..., w_i-1, among which only 0 means we do not lose bandwidth. Thus, the probability of wasting bandwidth is (w_j-1)/w_j = 1-1/w_j. The larger the window label is, the higher the probability of losing bandwidth at it. For example, if there are many leaves with window label 10, then there is a 90% probability of losing bandwidth at them. However, if the window label is 100, then the probability is 0.99. Therefore, we must try to prevent a leaf with a window label from splitting into a large number of leaves with the same large window label, and try to keep the average value of all the window labels in a tree as small as possible after each split operation, leaving leaves with small window labels open as long as possible.

However, minimizing the average value of window labels may cause more bandwidth loss. Thus, for each split operation, we must balance between the purpose of minimizing the average value of window labels and minimizing bandwidth loss at each segment. Bar-Noy's multilevel splitting greedy algorithm mentioned in Section 3.1 achieves part of this purpose. Consider, for instance, the case where K=2 and m=100. Segment S_1 will split the root node of C_0 into five leaves with window label 100, four leaves with window label 20, one leaf with window label 4 and one leaf with window label 2. The first leaf with window label 100 becomes the leaf with segment label 100 according to the multilevel splitting greedy algorithm as shown in Figure 3.7(a). The average window label is (5x100 + 4x20 + 1x4 + 1x2)/(5 + 4 + 1 + 1)=53.27. This tree structure is more flexible in minimizing bandwidth loss in subsequent assignments of leaves for segments than a tree structure formed after splitting the root node of C_0 into 100 leaves with period 100 by the greedy algorithm. In fact, we can pack 136 segments into this channel according to the multilevel splitting greedy algorithm rather than 100 segments according to the basic greedy algorithm.

However, the multilevel splitting greedy algorithm only works when $d_i = \lfloor r / w_i \rfloor$ in step 6 of Figure 3.4 is a composite number whose prime factors are all small. If d_i is a big prime number or has a big prime factor, then the multilevel splitting greedy algorithm will not prevent a leaf with the window label w_i from splitting into a large number of leaves with the same large window label $w_i{\bullet}d_i$. For example, for K=2 and m=100, segment S_2 will split C_1 into m+2-1=101 leaves with the same period 101 because 101 is a prime number (as shown in Figure 3.7(a)); for K=2 and m=141, segment S_2 will split C_1 into one leaf with period 2 and 71 leaves with period 141 because m+2-1=142=2x71

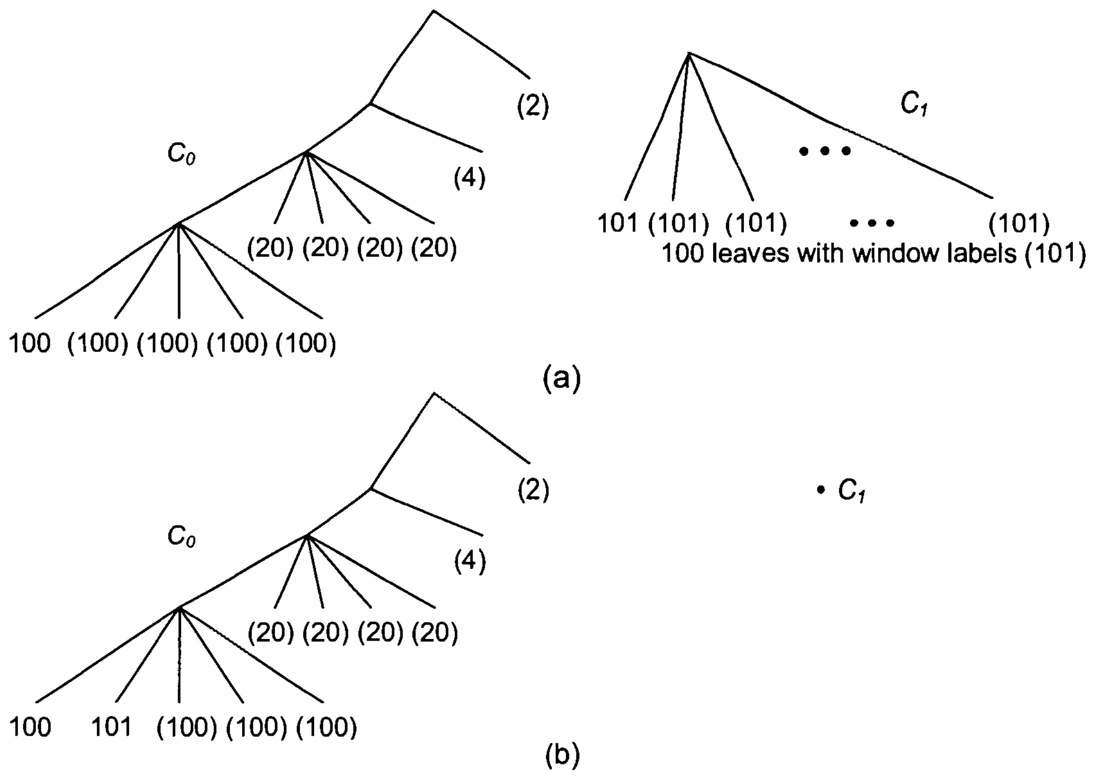


Figure 3.7 For K=2, m=100, the forest after two multilevel splits

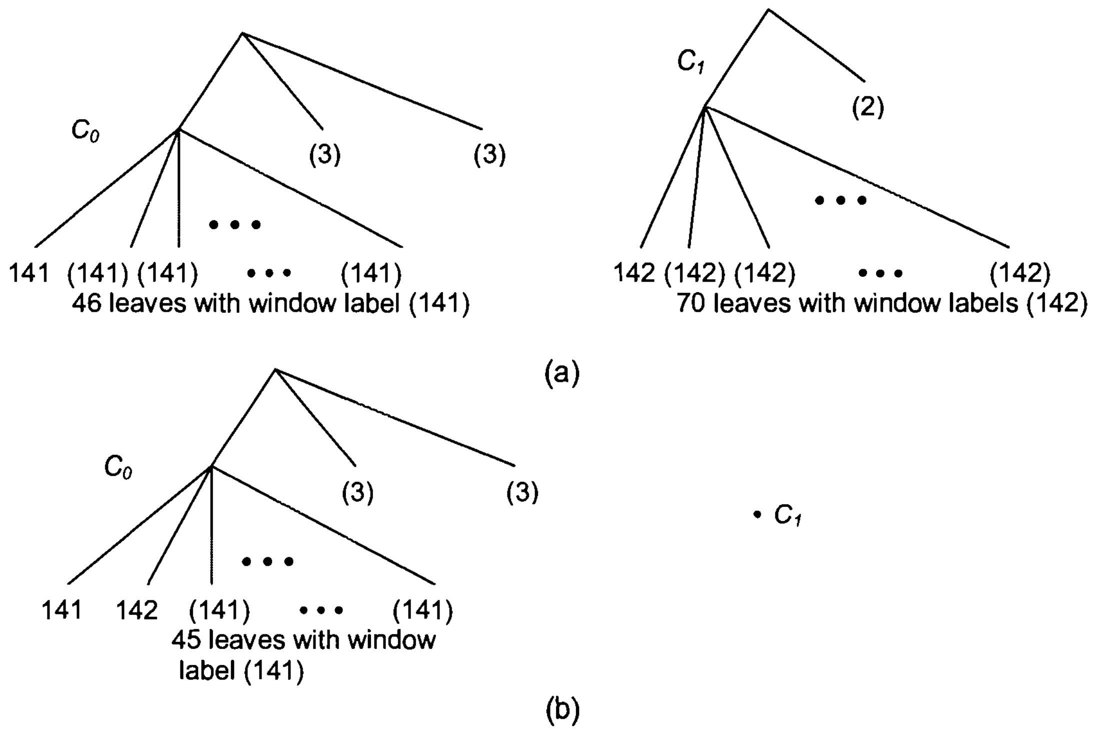


Figure 3.8 For K=2, m=141, the forest after two multilevel splits

and 71 is a big prime factor of 142 (as shown in Figure 3.8(a)). In Figures 3.7 and 3.8, we use the ideal period of each segment to indicate the segment label, so segment S_i is represented by an integer $m+i-1$. Each window label is represented as a number inside parentheses, where the number indicates the period of the window label. For example, (101) indicates a window label with period 101.

To further prevent a leaf with a window label from splitting into a large number of leaves with the same large window label (we call this split ***large split operation*** in the following), we try to give the lowest priority to a split operation in which the value of $d_i = \lfloor r / w_i \rfloor$ in step 6 of Figure 3.4 is a big prime number or has a big prime factor even if the corresponding $m_j =(r \bmod w_j)$ is very small. To do so, we introduce a parameter, *limiting_prime*, which is also a prime number, as a delimiting prime. For a split operation, if d_i is a prime number and bigger than a given *limiting_prime*, or the biggest prime factor of d_i is bigger than the given *limiting_prime*, then we will give a low priority to the split operation. In this way, we can control the structure of closed trees so that most of their internal vertices have no more than *limiting_prime* children. For each pair of K and m, the optimal *limiting_prime* that can maximize the number of packed segments may be different. By trial and error, we find that the optimal *limiting_prime* is in the range from 2 to 61 and too large a *limiting_prime* imposes little constraint to split operations. When *limiting_prime* is larger than 61, our results approach those of the multilevel splitting greedy algorithm. Therefore, we try *limiting_prime* values from 2 to 61 to find the optimal *limiting_prime* for each pair of K and m in our algorithm.

We have tried all the possible variant algorithms in order to choose the smallest $m_j = (r \bmod w_j)$ while considering the value of $d_i = \lfloor r / w_i \rfloor$ and its prime factors to minimize the probability of making a large split operation, but we find no variant can

```
Procedure: gfdpb_prime(K, m)
Input: K and m
Output: Value n, optimal limiting_prime and K closed trees, optimal_forest.
1) Initialize the number of segments max_seg=0 and optimal_limiting_prime = 0 and
   optimal_forest = NULL.
2) Assign 18 prime numbers between 2 and 62 inclusive, to the prime number array
   prime[0]=2, prime[1]=3, ..., prime[17]=61.
3) Initialize prime_index=0.
4) limiting_prime=prime[prime_index].
5) Initialize K channels as K singleton open trees with window labels 1 and initialize
   r=m since the ideal period of segment $S_1$ is m.
6) Let $w_1, w_2, \bullet\bullet\bullet, w_k$ be the ordered list of all the window labels in the forest in the
   increasing order of the corresponding channel numbers and then the start slot
   numbers. If this window label list is empty, go to step 16.
7) Initialize $i = 1$ for fetching the first window label in the above list.
8) Initialize the minimal remainder m_min=r, since $m_i$ in the next step is always less
   than r.
9) If $i \le k$, then $m_i = (r \bmod w_i)$, $d_i = \lfloor r / w_i \rfloor$ and $w_p=0$; otherwise, go to step 12.
10) If $m_i$ < m_min then {
        if $d_i$ is a prime number and m_min < r and $d_i$ > limiting_prime
            increment i and go to step 9.
        else
            m_min= $m_i$; $w_p = w_i$ }
    else if ($m_i$ == m_min and $w_i > w_p$) {m_min = $m_i$; $w_p = w_i$ }
11) Increment i by 1 and go to step 9.
12) Remove the window label $w_p$ from the window label list.
13) If$(d_p = \lfloor r / w_p \rfloor) = 1$, then replace the window label $w_p$ with the segment label r and go
    to step 15).
14) Calculate the prime factors of $d_p = \lfloor r / w_p \rfloor$ as $p_1, p_2, \cdots, p_j$. Apply the multilevel
    split operation to the window label $w_p$, following the increasing order of the prime
    factors of $d_p$ and attach the segment label r to the first leaf (the leftmost leaf) with the
    window label ($d_p \bullet w_p$) among the leaves just produced; then insert the rest of window
    labels into the window label list.
15) Increase r by 1 and go to step 6.
16) If(max_seg < (r- m)) { max_seg = r - m; optimal_limiting_prime = limiting_prime;
    optimal_forest=this_forest }.
17) If prime_index < 17 then increase prime_index by 1 and go to step 4; otherwise,
    output (max_seg, optimal_limiting_prime, optimal_forest)
```

Figure 3.9 Procedure *gfdpb_prime*

always outperform the others. Sometimes the difference among the variants is more than 1%. After trial and error, we finally settled on two procedures as shown in Figures 3.9 and 3.10. The two procedures have three loops inside them: the outer loop is from step 4 to 17 and loops through 18 *limiting_prime* values; the middle loop is from step 6 to 15 and is for calculating each *max_seg* which is the number of packed segments for a given *limiting_prime* value; the inner loop is from step 9 to 11 and is for finding the best window label to assign the given segment with the ideal period *r*. The better result of these two procedures (variants) for any given K and m is always within 1% of the best result from all the variants.

The two procedures comprise GFDPB. The first procedure *gfdpb_prime(K, m)* imposes a constraint on d_i whose value is a prime number and larger than a given *limiting_prime*, checking d_i only in the first condition of step 10 in Figure 3.9. The second procedure *gfdpb_prime_factor*(*K*, *m*) imposes a constraint on d_i whose biggest prime factor is larger than a given *limiting_prime*, checking d_i or d_p in all the three conditions of step 10 in Figure 3.10. Thus, we can see these two procedures are the two extremes for imposing constraints to prevent large split operations. The second procedure imposes the most severe constraint of all the variants we have tried on the d_i selection, while the first one imposes the least. In fact, more constraints on d_i selection sometimes causes less constraint on m_j selection and therefore causes more bandwidth loss. As a result, there is a trade-off between the selection of the best m_j and the selection of the best d_i. Thus, neither of these two procedures can always outperform the other. We try both of them and then pick the better one as the result of GFDPB in Figure 3.11.

For K=2 and m=100, both procedure *gfdpb_prime* and *gfdpb_prime_factor* replace the window label (100) next to the segment label 100 by segment label 101 in the C_0 tree, as shown in Figure 3.7(b), instead of splitting the root node of the C_1 tree

```
Procedure: gfdpb_prime_factor(K, m)
Input: K and m
Output: n, the optimal limiting_prime, and K closed trees
    1) to 9) are the same as procedure gfdpb_prime(K, m)
    10) if $m_i$ < m_min then {
            Calculate $d_i$'s biggest prime factor, biggest_prime_factor
            if biggest_prime_factor > limiting_prime and m_min < r
                    increment i and go to step 9
            else
                    m_min= $m_i$; $w_p$= $w_i$ }
        else if ($m_i$ == m_min and $w_i$ > $w_p$ ){
            Determine $d_i$'s biggest prime factor, biggest_prime_factor.
            if (biggest_prime_factor <= limiting_prime ) { m_min = $m_i$ and $w_p$= $w_i$ } }
        else if ($m_i$ > m_min) {
            $d_p = \lfloor r / w_p \rfloor$
            determine $d_p$'s biggest prime factor biggest_prime_factor_p
            if ( biggest_prime_factor_p > limiting_prime) { m_min= $m_i$; $w_p$= $w_i$ } }
11) Increment i by 1and go to step 9.
12) to 17) are the same as procedure gfdpb_prime(K, m)
```

Figure 3.10 Procedure *gfdpb_prim_factor*

```
Algorithm: GFDPB: gfdpb(K, m)
Input: number of channel K and m
Output: value n, K closed trees
(max_seg1, optimal_limiting_prime1, optimal_forest1) = gfdpb_prime(K, m)
(max_seg2, optimal_limiting_prime2, optimal_forest2) = gfdpb_prime_factor(K, m)
If (max_seg1 ≥ max_seg2)
    Output (max_seg1, optimal_forest1)
Else
    Output (max_seg2, optimal_forest2)
```

Figure 3.11 Algorithm for GFDPB scheme

into 101 leaves. This tree structure makes it more flexible to match the ideal periods of the subsequent segments. For K=2 and m=141, procedure *gfdpb_prime_factor* replaces the window label (141) next to the segment label 141 in the C_0 tree by segment label 142, as shown in Figure 3.8(b), instead of splitting C_1 into 71 leaves. However, according to procedure *gfdpb_prime,* for K=2 and m=141, segment 142 will split C_1 in the same way as in the multilevel splitting greedy algorithm, as shown in Figure 3.8(a).

3.3 Enhanced GFDPB

One problem with GFDPB is that it can't prevent the first segment with period m which is a prime or has a big prime factor, such as m=101 for K=1, from splitting the initialized singleton open tree into a large number of leaves with the same large window label. Therefore, we propose the Enhanced GFDPB (EGFDPB) algorithm in Figure 3.12. If m is a prime number and larger than 3, or if m is not a prime number but its biggest prime factor is larger than 10, then we reduce the period of the first segment from m to the nearest integer whose value is less than m and whose biggest prime factor is no larger than 10, and keep the ideal periods of the rest of segments the same as before. For a prime number m which is less than 10, we just decrement the period of the first segment from m to m-1.

First we must modify procedures *gfdpb_prime*(K, m) and *gfdpb_prime_factor*(K, m) into procedures *gfdpb_prime*(K, m, *first_per*) and *gfdpb_prime_factor*(K, m, *first_per*), respectively, where variable *first_per* is the modified period of the first segment. In the same way, procedure *gfdpb*(K, m) can be modified as *gfdpb*(K, m, *first_per*). Take *gfdpb*(2, 101, 101) =546, for instance. Since the ideal period of the first segment 101 is a big prime number, we reduce the period to 100. Therefore, *gfdpb*(2, 101, 101)=546 is modified as *gfdpb*(2, 101,100)=612 or *egfdpb*(2,101)=612. The number of packed segments is thus increased by 12%. From here we can see the benefit of preventing the large split operations.

```
Algorithm: EGFDPB: egfdpb(K, m)
Input: the number of channel K and m
Output: value n, optimal limiting_prime and K closed trees
  (max_seg1, optimal_forest1)=gfdpb(K, m, m)
  If (m is a prime number larger than 3, or m is not a prime number but the
  biggest prime factor of m is larger than 10) then {
          first_per = m;
          do{
                  first_per=first_per -1;
                  calculate the biggest prime factor of first_per;
          } while (first_per's biggest prime factor is larger than 10)
          (max_seg2, optimal_forest2)=gfdpb(K, m, first_per)
          if (max_seg1≥max_seg2)
                  return (max_seg1, optimal_forest1)
          else
                  return (max_seg2, optimal_forest2)
  }
  Else
          return (max_seg1, optimal_forest1)
```

Figure 3.12 Algorithm of Enhanced GFDPB

3.4 Performance Analysis of GFDPB and EGFDPB

To understand how well our GFDPB scheme performs with different numbers of channels and *m* values, we have calculated the values of *n* that result from GFDPB, EGFDPB, FDPB, RFS (or the greedy algorithm), and the multilevel splitting greedy algorithm. Here we apply RFS or the greedy algorithm under the fixed-delay policy. The results are shown in Figures 3.13 and 3.14, where the horizontal axis represents the value of *m* and the vertical axis represents the number of segments, *n*. The number in the name of each line represents the value of *K*. For example, fdpb_3 means FDPB with 3 channels, bar_3 means Bar-Noy's multilevel splitting greedy algorithm with 3 channels, and ub_3 means the upper bound for K=3. Here, the upper bound is calculated according to a small variation of Formula (2.6.1), in which $p(i)$ is changed to the ith prime number among the integers not less than m (e.g., if m=100 or m=101, then $p(1)$=101, $p(2)$=103, $p(3)$=107, etc) and "$f(i)=i-1$ if i is prime and $i \geq p(K)$" is changed to "$f(i)=i-1$ if i is prime and $i > p(K)$".

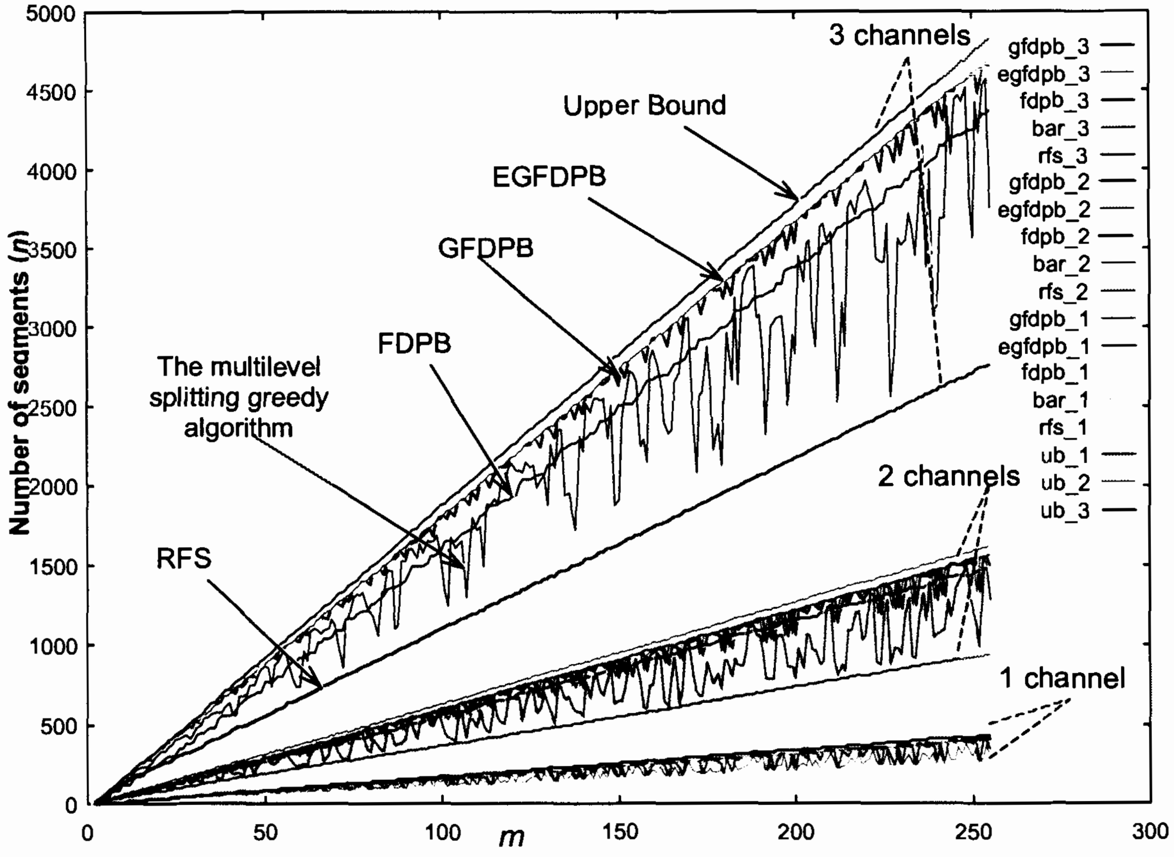


Figure 3.13 Diagram for K=1 to 3 and m=1 to 254

From these figures for each number of channels, we can see that the EGFDPB curve lies very close to and immediately below the upper bound curve. Note that the EGFDPB curve lies above the GFDPB curve when K<4, and almost completely overlaps the GFDPB curve when K>3. The GFDPB curve undulates slightly under the EGFDPB curve, whereas Bar-Noy's multilevel splitting greedy algorithm curve undulates very

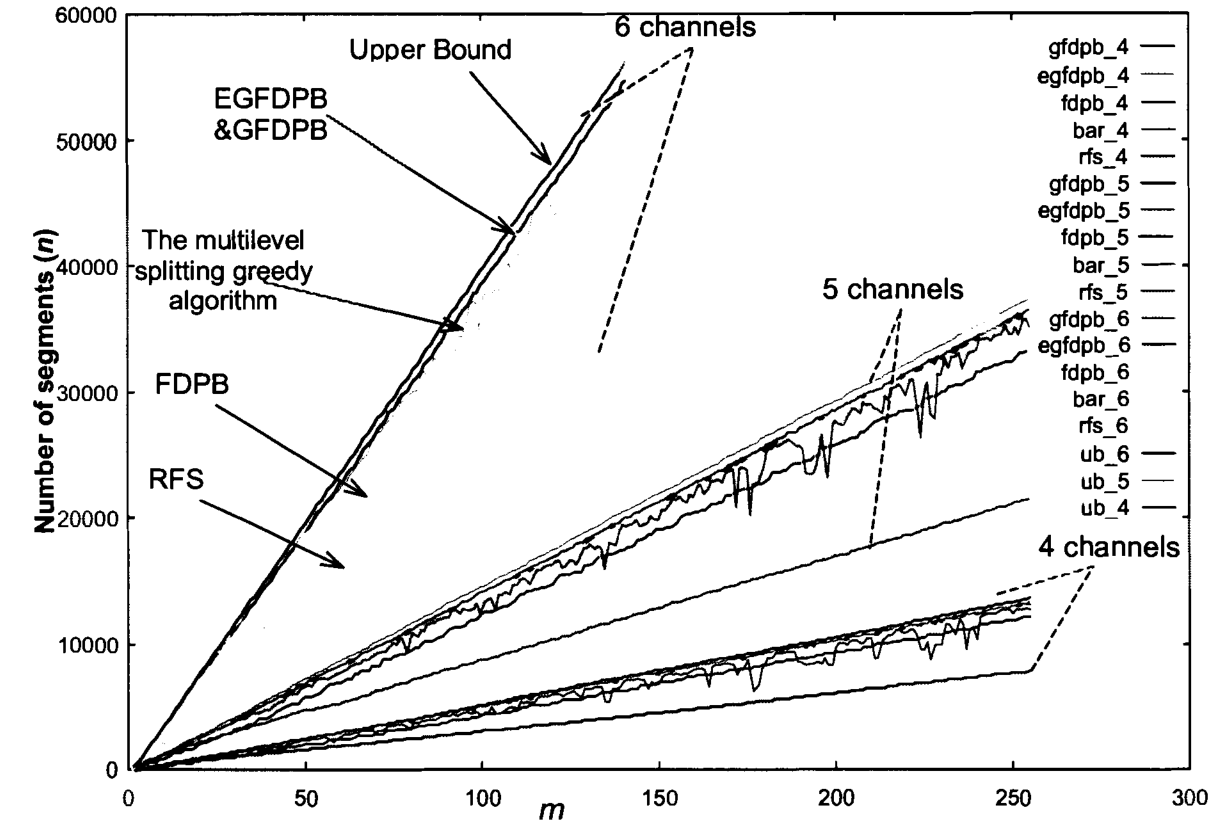


Figure 3.14 Diagram for K=4 to 6 and m=1 to 254

wildly below the GFDPB curve, and goes sometimes above and sometimes below the FDPB curve except for K=6. Some of the maximum points of curve bar_i with $1 \le i \le 6$ are rather close to curve gfdpb_i with $1 \le i \le 6$, but they are hard to be caught because of the irregularity and sparsity of this kind of maximum points. We can explain that the irregular wild undulations of curves bar_i with $1 \le i \le 6$ is due to their partial control of split operations since sometimes large split operations cause a heavy bandwidth loss. Surprisingly, we find FDPB is better than RFS in most cases unless m is very small. The reason is that the effect of the large split operations caused by RFS's one level splitting becomes bigger and bigger as m increases. Thus, RFS performs worse and worse with increasing *m*. The above reason can also explain why the derivative of the RFS curves decreases as *m* increases especially when *K*=6.

From these figures, we can see that of all the above schemes or algorithms, RFS or the greedy algorithm can pack the least number of segments, FDPB and Bar-Noy's multilevel splitting greedy can pack a moderate number of segments, and EGFDPB and GFDPB can pack the most number of segments. When more than three channels are used, the EGFDPB and GFDPB curves almost overlap, especially for *K*=6. For *K*=6, curve bar_6 is very close to curve gfdpb_6 or egfdpb_6 when m<50. When m≥50, curve bar_6 undulates below curve gfdpb_6 or egfdpb_6 and parts from curve gfdpb_6 and is progressively worse with increasing m.

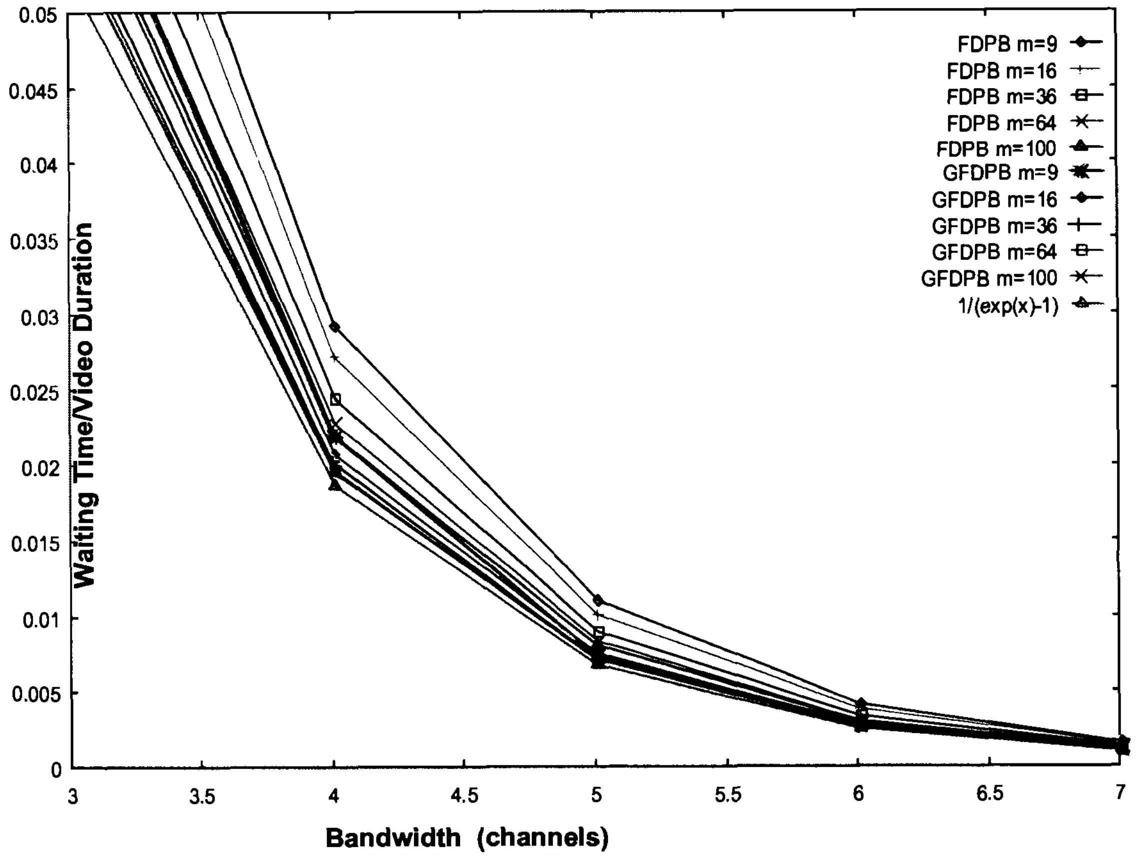


Figure 3.15 Waiting time for FDPB and GFDPB with different m

Given the number of channels K and the value of m, the inverse of n offered by each scheme reflects the maximum waiting time before a client can start viewing a video. Figure 3.15 shows the normalized waiting time (m/n) achieved by GFDPB and FDPB for m=9, 16, 36, 64, and 100. All the bandwidths are expressed as multiples of the video consumption rate b, and all the waiting times are expressed as fractions of the video duration D. The top five curves in the figure represent the results of FDPB and the curve at the bottom represents the lower bound on FDPB, i.e., $1/(e^K - 1)$. We can see all the normalized waiting time values achieved by GFDPB are much smaller than those of FDPB and approach the lower bound very quickly with increasing m.

BW/*b*	3	4	5	6	7
GFDPB *m*=100	0.05506608	0.01955034	0.00705219	0.00256937	9.399644E-4
GEBB *n*=100	0.05488886	0.02020000	0.00766276	0.00295594	1.153780E-3
GEBB *n*=500	0.05289245	0.01896235	0.00695542	0.00257550	9.581609E-4
PHB *m*=8	0.0556	0.0199	0.00723	0.002648	9.72E-4
PHB *m*=100	0.0526500	0.01875	0.006818	0.0024974	9.173E-4

Table 3.1 Comparison of GFDPB, GEBB and PHB

Table 3.1 shows the normalized waiting time achieved by GFDPB, GEBB and PHB. The bandwidth is expressed as multiples of the video consumption rate b. From the table, we can see GFDPB with the number of channels K>3 and m=100 achieves much shorter normalized waiting time than GEBB with the same total bandwidth and 100 channels. GFDPB with K>5 and m=100 achieves even shorter normalized waiting time than GEBB with the same total bandwidth and 500 channels. GFDPB with m=100 achieves shorter waiting time than PHB with m=8, where a video is broadcast in many more channels. For example, given the bandwidth B=6b, PHB with m=8 needs to use 3021 channels to achieve a normalized waiting time of 0.002648, which is still longer than that achieved by GFDPB with six channels. At the cost of an enormous number of channels, PHB with m=100 achieves shorter waiting time than GFDPB. For example, given bandwidth B=6b, PHB achieves normalized waiting time of 0.0024974, which is 2.8% shorter than that achieved by GFDPB with m=100, by partitioning a video into 40041 segments and broadcasting each of them on a separate channel with decreasing bandwidth. However, to handle so many channels with decreasing bandwidth is likely to be a daunting task. Therefore, Table 3.1 shows that our GFDPB scheme is better than GEBB and PHB with the same total bandwidth and many more channels. Only if the number of channels used by GEBB and PHB increases to a great extent, the GEBB and PHB schemes can outperform GFDPB with the same total bandwidth and a much smaller number of channels. However, compared with GEBB, GFDPB needs to pay the price of handling many more segments than GEBB. For example, for K=6, each video will be partitioned into 38920 segments in GFDPB with m=100, but only 540 segments in GEBB. Thus, the overhead of GFDPB is much more than that of GEBB.

# of channels	4	5	6	7	8	9	10	11	12
Upper bound	28	80	220	604	1650	4501	12260	33369	90806
Bar-Noy Best bound	28	77	211	570	1573	4325	11759	31677	86428
Bar	25	73	203	561	1563	4325	11759	32065	87329
RFS	25	73	201	565	1522	4284	11637	31677	86428
GFDPB	25	73	205	566	1567	4328	11893	32467	88357

Table 3.2 Number of the packed segments for m = 1

Table 3.2 compares the numbers of the packed segments achieved by GFDPB for m=1 with those achieved by RFS [59], the multilevel splitting greedy algorithm (indicated by "Bar"), and Bar-Noy's best bound [58]. Bar-Noy's best bounds for small numbers of channels (K=4, 5, 6) were achieved by hand tuning the results of the greedy algorithm (RFS) and realized by non-perfect schedules. Bar-Noy's best bounds for 7 to 10 channels were obtained from their modifications of the basic greedy algorithm, which were tuned according to each specific number of channels and use ad-hoc methods. They are the best known schedules for *m*=1. The upper bounds are calculated according to Theorem 2 of [59], as shown in formula (2.5.1). We can see from Table 3.2 that even for *m*=1, GFDPB outperforms RFS and the multilevel splitting greedy algorithm ("Bar"), and is better than Bar-Noy's best bound if the number of channels is bigger than 8.

Channel	FDPB	RFS	Bar	GFDPB	UB
1	12	9	12	12	14
2	42	37	44	47	53
3	116	121	140	143	159
4	292	377	405	409	449
5	770	1086	1146	1163	1240
6	2046	3055	3205	3219	3393
7	5477	8307	8725	8860	9249

Table 3.3 Number of the packed segments for *m* = 9

Channel	FDPB	RFS	Bar	GFDPB	UB
1	156	100	136	158	170
2	565	370	452	606	634
3	1650	1105	1335	1816	1895
4	4563	3168	4778	5115	5324
5	12418	8809	13420	14180	14646
6	33684	25432	37717	38920	39986
7	91321	72029	104478	106387	108872

Table 3.4 Number of the packed segments for *m* =100

In Tables 3.3 and 3.4 we compare the number of packed segments achieved by GFDPB with those offered by FDPB, RFS, and the multilevel splitting greedy algorithm (indicated by "Bar") in the case of m=9 and m=100. After computing the average improvement, we find that GFDPB is on average 34.96% better than FDPB for m=9, and 11% better than FDPB for m=100, and on average 1.92% better than the result of the multilevel splitting greedy algorithm for m=9, and 14.86% better than that of the multilevel splitting greedy algorithm for m=100. In Figures 3.13 and 3.14, we can see that when m is small, the result of the multilevel splitting greedy algorithm is generally close to the GFDPB result and is much worse than the GFDPB result when m is big.

In addition to our above analysis, we also know that FDPB [57] and RFS [59] are better than all other Pagoda schemes in the third group of broadcasting protocols, and PHB and GEBB outperform all other schemes in the first and the second group. Therefore, we can conclude that EGFDPB or GFDPB outperforms all other schemes in the third group, and they achieve the lowest maximum waiting time of all currently known protocols given the same server bandwidth and the same number of channels. Sometimes, the results of the multilevel splitting greedy algorithm are very close to GFDPB, especially when m is small, but its performance is very unstable and uncontrollable. Furthermore, when $K < 4$, the maximum waiting times achieved by the multilevel splitting greedy algorithm are longer than those offered by FDPB in most cases. The result of GFDPB with K<4 is slightly unstable when m is a big prime number or has a big prime factor, but it becomes more stable for K>3. We are able to choose a non-prime number m whose biggest prime factor is no larger than 10 so that we can only use GFDPB to find satisfying results quickly.

3.5 Server Multiplexing Scheme

In this section, we will analyze the server multiplexing scheme for GFDPB, which can be applied to FDPB and RFS as well. Intuitively, after getting a broadcasting schedule from GFDPB, FDPB or RFS, we can form a channel schedule for each channel as mentioned in Section 3.1. The channel schedule is a sequence of segments to be transmitted on the corresponding channel, starting from time slot 0, and has a length equal to the least common multiple (LCM) of the actual broadcasting periods (in slots) of all the segments in the channel. This channel schedule can be put into a circular array, so this array will contain a sequence of segments that is repeated infinitely in the channel. Each indexed position in the circular array corresponds to a time slot. We can keep a pointer for each channel schedule array and initially the pointer points to the first segment of the array. Every time we fetch the K segments that are pointed by the K pointers, send them to the corresponding broadcasting channels, and then let each of the K pointers point to the next segment of each array. In this way, the K pointers follow the K circular arrays forever.

During simulation, we found a serious problem with the above intuitive server multiplexing scheme. If the number of segments of each video exceeds 300, the LCM of the actual periods of all the segments broadcast in one channel may be bigger than 1 million. If the number of segments exceeds 2000, then the LCM will be larger than 1 billion. If the number of segments is much bigger than 2000, then the LCM may be out of range of four-byte integers. Therefore, it takes up too much memory to be a practical algorithm.

From Figures 3.1 and 3.2, we can see that the start slot of a segment is normally less than the segment's actual period minus 1. For example, the start slot of $SS(C_i, 0, 12, A)$ is slot 0 and its period is 12. The start slot of $SS(C_i, 3, 12, F)$ is slot 3 and its period is 12. Only the start slot of the rightmost leaf SS(C_i, 5, 6, H) is equal to its period minus 1, i.e. 5. According to this observation, we have the following lemma.

Lemma 3.1 In a closed tree, a segment has an offset equal to its actual period minus 1 if and only if it is represented by the rightmost leaf such that each of its ancestor nodes is the rightmost among its sibling nodes.

Proof: We use the symbols introduced in Formulae 3.1.1 and 3.1.2 in Section 3.1. For a leaf node J_m, its ancestors are named J_{m-1}, J_{m-2}, ..., J_1, J_0, where J_0 is the root node and J_{i-1} is the parent of J_i ($0< i \le m$). Then, Formulae 3.1.1 and 3.1.2 are changed to be $T(J_i) = T(J_{i-1}) \bullet d(J_{i-1})$ and $a(J_i) = a(J_{i-1}) + h(J_i)\, T(J_{i-1})$, where $T(J_i)$ is the period of J_i and $a(J_i)$ is the start slot of J_i. We have

$$a(J_m) = a(J_{m-1}) + h(J_m)T(J_{m-1}) = a(J_{m-2}) + h(J_{m-1})T(J_{m-2}) + h(J_m)T(J_{m-1}) = \bullet\bullet\bullet$$
$$= a(J_0) + \sum_{i=1}^{m} h(J_i)T(J_{i-1})$$

Since $h(J_i) \le (d(J_{i-1})-1)$, we have

$$a(J_m) \le a(J_0) + \sum_{i=1}^{m}(d(J_{i-1})-1)\bullet T(J_{i-1}) = a(J_0) + \sum_{i=1}^{m-1}(d(J_{i-1})-1)\bullet T(J_{i-1}) + T(J_m) - T(J_{m-1})$$
$$= a(J_0) + \sum_{i=1}^{m-1}(d(J_{i-1})-1)\bullet T(J_{i-1}) + (T(J_{m-1}) - T(J_{m-2})) + (T(J_m) - T(J_{m-1}))$$
$$= a(J_0) + \sum_{i=1}^{m-1}(d(J_{i-1})-1)\bullet T(J_{i-1}) + (T(J_m) - T(J_{m-2})) = \bullet\bullet\bullet$$
$$= a(J_0) + (T(J_1) - T(J_0)) + (T(J_m) - T(J_1)) = a(J_0) + T(J_m) - T(J_0) = T(J_m) - 1$$

The equality happens if and only if $h(J_i)=d(J_{i-1})-1$, where $0< i \le m$. ■

To solve the problem caused by the tremendous memory requirement, we give up on the above server multiplexing scheme, which uses channel schedules. For each channel, we make a circular array and the size of the array is equal to the maximum broadcasting period of all the segments broadcast on the channel. From the channel, we fetch a sequence of segments starting from time slot 0 with the array size to be transmitted on the channel, and put the sequence of segments into the circular array.

Lemma 3.1 guarantees that the array will contain all the segments broadcast in the corresponding channel at any time. Each time slot corresponds to an indexed position of the array, which contains the address of a segment and the corresponding period. Assume the array size is Δ. At the beginning of time slot α, we will fetch the address of a segment from index (α mod Δ) of the array. We set a pointer for each array, initially pointing to index 0 of the array. We must have K senders to send segments to the K channels simultaneously.

Each sender works as follows:

> For each time slot, slot i, first, the sender fetches the address of a segment with period p from the indexed position pointed to by the corresponding pointer. Second, go to the address to fetch the segment and send it to the corresponding channel. Third, rewrite the position of index $(i+p) \bmod \Delta$ with this segment's address and period. Finally, let the pointer point to the next indexed position, index $(i+1) \bmod \Delta$, of the corresponding circular array.

In the way described above, we constantly update the K circular arrays to guarantee that the interval between any two consecutive broadcasting of a segment is exactly equal to its actual broadcasting period calculated by GFDPB, RFS, or FDPB. For K=6 and m=100, we have the number of segments n=38920, achieved by GFDPB, so the size of the six circular array is less than 38920 X 6 = 233520.

Figure 3.16(a) shows the first channel schedule for the case of K=2 and m=3. The maximum broadcasting period in this channel is 9, so we use a circular array with size 9 and initialize the array as shown in Figure 3.16(b). Here, for simplicity, we assume the address of each segment is equal to its segment number. Thus, segment S_i is represented as *i.(actual period)*. For example, 1.3 represents the address of segment S_1 with the actual broadcasting period 3. In Figure 3.16(b) the index 0 of the array is at the top and filled with 1.3. The array index is increased clockwise to index 9, which is filled with 9.9. Take time slots 0 to 4 for instance. In slot 0, the sender sends segment S_1

indicated in index 0 and then rewrites index (0+3) mod 9=3 with 1.3. In slot 1, the sender sends S_4 indicated in index 1 and then rewrites index (1+6) mod 9=7 with 4.6. In slot 2, the sender sends S_7 indicated in index 2 and then rewrites index (2+9) mod 9=2 with 7.9. In slot 3, the sender sends S_1 indicated in index 3 and then rewrites index (3+3) mod 9=6 with 1.3. In slot 4, the sender sends S_5 indicated in index 4 and then rewrites index (4+6) mod 9=1, where we replace 4.6 with 5.6.

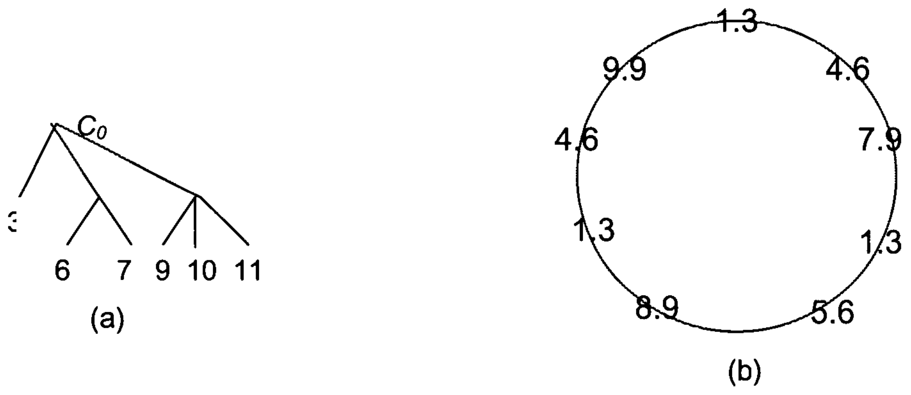


Figure 3.16 Circular array for server multiplexing

3.6 Client Demultiplexing Scheme

In this section, we will discuss the demultiplexing scheme on the client side. Since we need to download from all the channels simultaneously, we must have *K* loaders to download simultaneously from *K* channels and one displayer to display the video at the same time. First, we have the following two observations:

- We need to know which segments are available in each time slot so that they can be downloaded and saved in order. Thus, every segment must bear a segment number or a sequence number in its header. For *K*=6 and *m*=100, a 2-byte header is enough for identifying segments less than 40,000. In the following, we assume the sequence number of a segment is equal to its segment number.

- If we start downloading a segment from the middle, we don't know which segment it is; therefore, we let each user wait for the beginning of a new slot, and then download for $m-1$ time slots in advance. After that, the user starts viewing the movie.

According to the second observation, at the beginning of the $(m+i-1)$-th slot, clients can always display the i-th segment from the buffer or display directly from one of the loaders. Thus, we can start displaying S_1 at the beginning of the m-th slot, and then S_2 at the beginning of the $(m+1)$-th slot, and so on.

If we have plenty of memory or buffer space, we can allocate continuous space for the whole movie, and each segment address can be calculated by a formula according to the segment size and the starting address of the first segment. At the beginning of each time slot, each loader checks the segment header first, calculates the address, and then downloads the segment to the address. After each loader has preloaded $m-1$ segments, the displayer starts displaying the video from the beginning of the continuous buffer space, and then goes through the buffer space until reaching the last segment of the video.

If we want to save buffering space, we need an ***address array*** with the size of the total number of segments of a video to hold the addresses of all the downloaded segments. Also, we need a free list to hold all the free memory space. We present our demultiplexing scheme which we call the loader scheme next.

Each loader works as follows (the ***loader scheme***):

At the beginning of each time slot, the loader downloads a segment header and compares its sequence number i with the sequence number j of the segment just starting to display. If $i < j$, the loader discards the segment with sequence number i (segment i) in the present time slot. If $i=j$ and the displayer asks for it from loaders, the loader directly sends segment i to the displayer. If $i=j$ and the displayer doesn't ask for it from loaders, the loader discards the segment. If $i > j$, the loader checks the address array for the address of segment i and does as follows. If the address for segment i in the present time slot is found, it means segment i has been downloaded and the loader does not need to download it again; otherwise, the loader checks the free list for a free block to download segment i and writes the address of segment i to the address array.

The displayer works as follows:

Once $m-1$ time slots have elapsed after the moment each loader starts downloading segments, the displayer starts to fetch segment addresses from the address array in order and displays the segments one by one. If an address is not found in the address array, the displayer checks the K loaders to find the one that is downloading the segment, and then reads directly from the loader. Finally, the displayer places the address of the segment just displayed to the free list and starts to fetch the next segment.

The above loader scheme takes two steps to check the segments that have been downloaded but have not yet been displayed: first, compare i and j; second, check the address array. We can save time by adding a bit vector with its size equal to the total number of segments. Each bit represents a segment, and it is 1 if this segment has been downloaded or displayed; otherwise, it is 0. For m=100 and K=6, this bit vector only needs 40 Kbits = 5 Kbytes of memory. Therefore, the loader scheme is modified as follows.

Each revised loader works as follows:

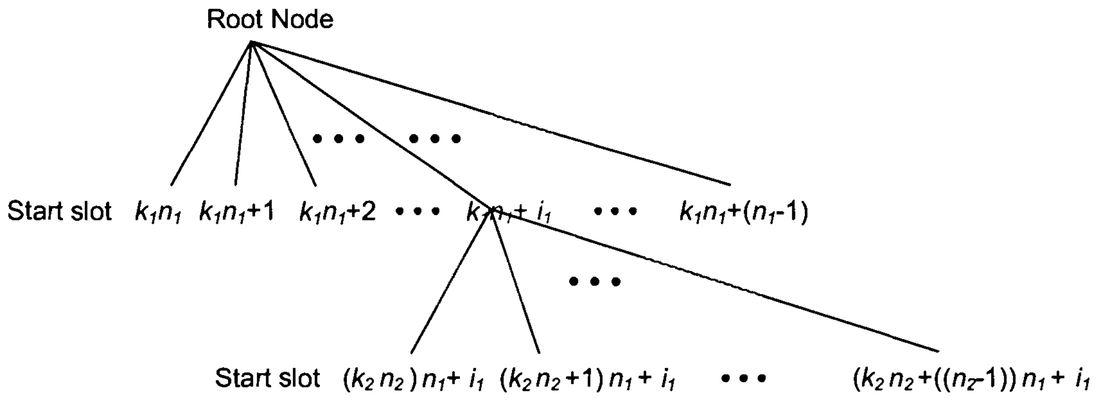
At the beginning of each time slot, the loader downloads a segment header and checks the bit vector to see whether the segment has been downloaded. If the bit is 1, the loader quits downloading in this time slot. If the bit is 0 and the segment is just needed for display, the loader sends the segment directly to the displayer and sets the bit to 1. If the bit is 0 and the segment does not have to be displayed immediately, the loader checks the free list for a free block to download the segment, changes the corresponding bit in the bit vector to 1, and then writes the address in the address array.

Many computers support very efficient bit-manipulation instructions in assembly languages. Often high level program languages also provide these kind of instructions.

Thus, we can make a decision about whether to download the segment in the present time slot very quickly in the above scheme. The cost is only a small multiple of K bytes of memory.

3.7 Channel Schedule Formats Translation

The translation from the tree representation to the (slot) sequence representation was described in Section 3.1. The method first copies the channel number and the segment number, and then calculates the period and the start slot (offset) of each segment label of the tree. Here, the segment number is equal to the segment's ideal period.


Root Node
Start slot k_1n_1 k_1n_1+1 k_1n_1+2 • • • k_1n_1+ i_1 • • • k_1n_1+(n_1-1)
• • •
Start slot (k_2n_2)n_1+ i_1 (k_2n_2+1)n_1+ i_1 • • • (k_2n_2+((n_2-1))n_1+ i_1
Figure 3.17 Start slot pattern of the round robin tree

In this section, we discuss the translation from the (slot) sequence representation to the tree representation of a channel schedule. According to the recursive round-robin character of the tree representation shown in Figure 3.3, for a tree whose root node has n_1 children labeled with the numbers 0, 1, ..., n_1−1, the start slot (offset) of each segment label that is a descendant of child i_1 ($0 \le i_1 < n_1$) of the root node can be expressed as $k_1n_1 + i_1$, where k_1 is a non-negative integer and $0 \le i_1 < n_1$. If $k_1n_1 + i_1$ is the start slot (offset) of a segment label in the tree shown in Figure 3.17, k_1 is the start slot (offset) of the segment label in the subtree of child i_1 ($0 \le i_1 < n_1$) of the root node, and $k_1 = k_2n_2 + i_2$. Integer n_2 is the number of the children of child i_1 of the root node, $0 \le i_2 < n_2$, and k_2 is a non-negative integer, as shown in Figure 3.17. According to this observation, if a segment whose start slot (offset) is s in a channel schedule tree, this segment should be a descendant of child c ($c = s \bmod n_1$) of the root node, and $\lfloor s/n_1 \rfloor$ should be the start

```
Algorithm: Translate a channel schedule from the slot sequence representation to
  the tree representation
Input: A channel schedule in a slot sequence list
Output: A channel schedule in the tree representation
  1) Sort the slot sequence list in the increasing order of the ideal periods.
  2) Pick the slot sequence, whose start slot, actual period and ideal period are
     indicated as (startSlot, actualPeriod, idealPeriod), from the head of the list and
     remove the slot sequence from the list.
  3) If the present segment is the first segment of the channel, apply the multilevel
     split operation to the root node following the increasing order of the prime factors
     of the actual period of the first segment, and then attach the segment label,
     idealPeriod, to the leftmost leaf. Then go to step 2).
  4) If the present slot sequence is not the first, use startSlot to find an empty leaf.
     Initialize s=startSlot, r=0, and $J=J_0$, where J is the current node and $J_0$ is the root
     node.
  5) Calculate $r = s \bmod d(J)$ and $s = \lfloor s/d(J) \rfloor$, where d(J) is the number of
     children of node J.
  6) Update J as child r of itself. If J becomes a leaf with a window label w, apply
     multilevel split operation to J, following the increasing order of the prime factors of
     integer actualPeriod/w, and then attach segment label idealPeriod to the leftmost
     leaf of the subtree of node J; otherwise, go to step 5).
  7) If the slot sequence list is not empty, go to step 2); otherwise, output the tree.
```

Figure 3.18 The channel schedule format translation

slot (offset) of this segment in the subtree of child c. We can recursively apply the same rule to the subtree of child c of the root node. According to the above description, we have our translation algorithm in Figure 3.18. Since segments are placed in a tree in the increasing order of the corresponding ideal periods in the GFDPB algorithm shown in Figures 3.9 to 3.11, we also place slot sequences, each of which corresponds to a segment, to a tree in the increasing order of the corresponding ideal periods. Therefore, we sort the slot sequence list in step 1. The algorithm in Figure 3.18 has two loops inside it: the outer loop is from steps 2 to 7 and loops through the slot sequence list; the inner loop is from steps 5 to 6 and is for placing a given slot sequence, which is not the first slot sequence in the list, to the tree.

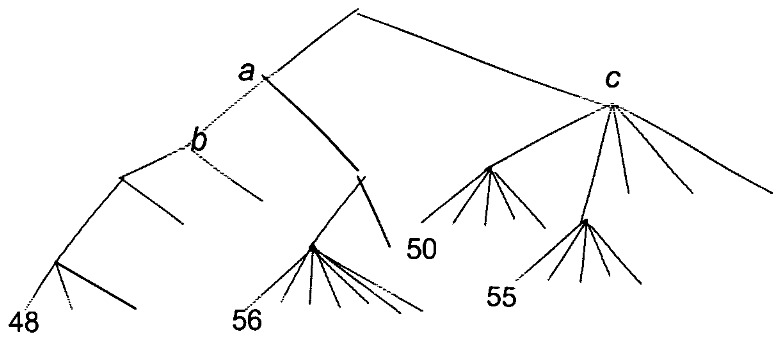


Figure 3.19 Example for the schedule format translation

For example, let the first four segments in the order of their ideal periods be SS(0, 0, 48, 48), SS(0, 1, 50, 50), SS(0,3,50,55), and SS(0,2,56,56). The first, the second, the third, and the fourth integer inside each pair of brackets represent the corresponding segment's channel number, start slot, actual period and ideal period, respectively. First, we need to place SS(0, 0, 48, 48). According to step 3 of Figure 3.18, we apply the multi-level split operation to the root node following the order of 2, 2, 2, 2 and 3 since 48=2x2x2x2x3, as shown in Figure 3.19. Second, we place SS(0,1,50,50). Since the root node has two children and 1 mod 2=1, segment label 50 is a descendant of child 1 of the root, which is an open leaf. We apply the multi-level split operation to the leaf following the order of 5 and 5 since 50/2=25=5x5. Third, we place SS(0,3,50,55). In step 4 of Figure 3.18, we have *s*=3, *r*=0, and *J*=root. In the first loop of steps 5 and 6, *r*=3 mod 2=1, *s*=$\lfloor 3/2 \rfloor = 1$, and *J*=*c*. In the second loop, node *c* has five children, so *r*=1 mod 5=1. Since child 1 of node *c* is an open leaf and has a window label 10, we split the leaf into five child leaves since 50/10=5. Similarly, we place SS(0,2,56,56). Figure 3.19 only shows the segment labels, and the window labels are left empty.

Chapter Four
Group-Based Broadcasting Schemes

As before, we consider a video of duration D to be broadcast over K channels $\{C_i \mid 0 \le i < K\}$. The bandwidth of each channel is equal to the video consumption rate b, so that the total bandwidth is equal to Kb. In the model adopted in the harmonic windows scheduling, the number of pages n that can be packed in the K channels must satisfy

$$H(n) = \frac{1}{1} + \frac{1}{2} + \bullet\bullet\bullet + \frac{1}{n} \le K$$

This implies that there is bandwidth equal to $K - H(n)$ that is not utilized if this quantity is non-zero. In this chapter, we try to make use of this "wasted" bandwidth to pack more pages into K channels. To this end, we first describe the *Harmonic Group Window Scheduling* (HGWS) problem in Section 4.1. Then, in Section 4.2, we present the *Harmonic Page-set Broadcasting* (HPB) scheme as a solution to HGWS. In Section 4.3, we simulate HPB and analyze its results. Finally, in Section 4.4, we propose the *Preloading Page-Set Broadcasting* (PPSB) scheme to remedy HPB's shortcoming with respect to maximum waiting time.

4.1 Harmonic Group Window Scheduling (HGWS) Problem

The HGWS was introduced in [67], which presents many basic properties of the problem. The HGWS problem has two features different from the *Fixed-Length Segment-scheduling (FLSS) Problem* defined in [59] for the common features of all other broadcasting schemes in the third group. First, HGWS groups consecutive slots into ***blocks***, and second, it groups consecutive pages into ***page-sets***.

Each video is partitioned into N ***pages*** of duration $d=D/N$, where a page is an equal-sized segment. A page is the basic unit of the transmission of a video in the HGWS problem. Each of these N pages is broadcast at a certain interval over the K channels such that each page transmission occupies a time ***slot*** of duration d in some channel. We group β consecutive time slots into one time ***block***, as shown in Figure 4.1 for β=5. Integer β is called the ***block size***. We label time slots from 0 to β-1 inside each time block. Each page-set consists of a number of consecutive pages. All the pages in page-set i (the first page-set is numbered 1) are broadcast exactly once every i blocks so that all of them have the same broadcasting period, i.e., i time blocks or, equivalently, $i\beta$ time slots.

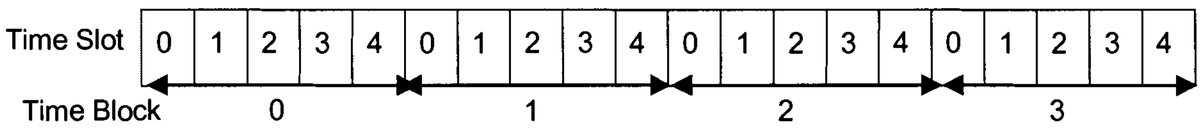


Figure 4.1 Block diagram for β = 5

A client waits until the beginning of a new time block to start watching a video and at the same time downloading simultaneously from all K channels. Channel C_0 continuously repeats the first page-set (page-set 1), consisting of page 1 to page β, to ensure that they are repeated in every time block. For example, in Figure 4.1, for j=0, 1, 2, 3, 4, slot j is allocated to page j+1. This way, once a client starts viewing page 1, pages 2 to 5 will be available for display just in time. It is easy to see that the maximum waiting time and average waiting time are βd (one block duration) and $0.5\beta d$ (half block duration), respectively.

Grouping every β slots as a block is similar to partitioning a channel into β ***subchannels*** in Fixed Delay Pagoda Broadcasting (FDPB) [57]. Slot 0 of every block belongs to subchannel 0, and slot 1 of every block belongs to subchannel 1, and so on.

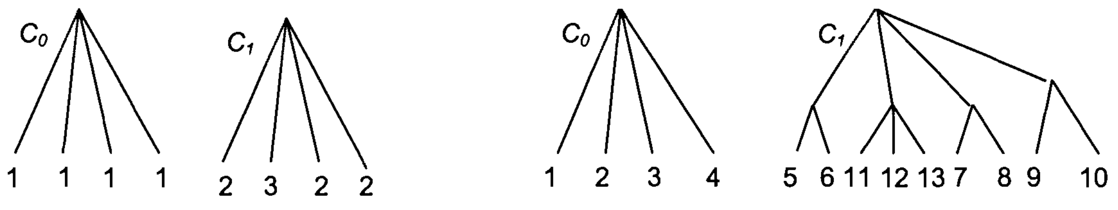
In general, slot i ($0 \leq i \leq \beta-1$) of every block belongs to subchannel i. As we saw above, the β subchannels of channel C_0 are allocated to the β pages of page-set 1.

Unlike FDPB which maps pages into subchannels in a strictly sequential fashion, we first assign each page-set to several (possibly inconsecutive) subchannels, observing the continuous display constraint that we will discuss later. This assignment produces a ***page-set schedule***, then we map pages into page-sets in a strictly sequential fashion to get a ***page schedule***. If subchannel j is allocated to page-set i, we map only i consecutive pages of page-set i to subchannel j, since each page in page-set i must be broadcast once in every i blocks. As stated above, we map the first β pages into the subchannels allocated to page-set 1, and then the next β or more pages into those of page-set 2, and so on. Subchannels belonging to different channels can be allocated to a page-set. We always map consecutive pages into a page-set in the order of its allocated subchannel numbers.

We can use a forest consisting of K round-robin trees to represent a page-set schedule in the tree representation mentioned in Section 3.1. One tree is assigned for each channel, as shown in Figure 4.2(a) for K=2 and β=4. Each tree has β leaves representing the β subchannels of the corresponding channel. If we number the β leaves of each tree 0, 1, ..., $\beta-1$, from left to right, leaf i corresponds to subchannel i or slot i of each block, where $0 \leq i \leq \beta-1$. Each leaf in the tree is labeled with a page-set number called *page-set label,* representing the page-set the corresponding subchannel is allocated to.

A page schedule further splits each leaf with a page-set label in a page-set schedule into level 2 leaves representing individual pages, except the leaf with page-set label 1. For example, in Figure 4.2, pages 5 and 6 are mapped into page-set 2 and subchannel 0 of C_1 is allocated to them, so we split the leftmost leaf of C_1 in (a) into two leaves at level 2 with page labels 5 and 6, as shown in (b). According to the algorithm in Figure 3.3, we can easily translate each tree of a page schedule to a channel schedule to be broadcast repeatedly on the corresponding channel.

Figure 4.2 shows a complete example. For $\beta = 4$ and $K = 2$, page-set 1 is assigned to the four subchannels of channel C_0, and page-set 2 is assigned to subchannels 0, 2 and 3 of C_1, and page-set 3 is assigned to subchannel 1 of C_1. We thus get a page-set schedule shown in Figure 4.2(a). We will discuss in detail why we assign page-sets to subchannels in this way in the next section. The purpose is to guarantee continuous display of a video and to pack as many pages as possible into the given channels. In the following steps, we map pages into page-sets to get the corresponding page schedule as shown in Figure 4.2(b). First, we map pages 1, 2, 3, 4 to page-set 1 and allocate the four subchannels of channel C_0 to the pages. Then, we map pages 5, 6, 7, 8, 9, 10 to page-set 2, since there are three subchannels allocated to page-set 2 in the page-set schedule shown in Figure 4.2(a) and each subchannel can broadcast two pages of page-set 2. Now, pages 5, 6 are mapped into subchannel 0 of C_1, pages 7, 8 into subchannel 2 of C_1, and pages 9, 10 into subchannel 3 of C_1. Finally, since there is only one subchannel allocated to page-set 3 in the page-set schedule shown in (a), and each subchannel can broadcast three pages of page-set 3, we map pages 11, 12, 13 to page-set 3 and allocate subchannel 1 of C_1 to the pages. Figure 4.2(b) shows the above page schedule. Each label in Figure 4.2(b) represents a page number called a *page label*. Applying Procedure Tree-to-Schedule in Figure 3.3 to the four round-robin subtrees of channel C_1 in Figure 4.2(b) yields the channel schedules of four subchannels of C_1, <5, 6>, <11, 12, 13>, <7, 8>, and <9, 10>, respectively. Following the procedure in Figure 3.3, we can get the channel schedule of C_1, <5, 11, 7, 9, 6, 12, 8, 10, 5, 13, 7, 9, 6, 11, 8, 10, 5, 12, 7, 9, 6, 13, 8, 10>, which is broadcast repeatedly on channel C_1. Figure 4.2(c) shows the first five blocks of the broadcasting sequence of pages of C_0 and C_1, and (d) shows the broadcasting sequences of the four subchannels of C_1. For example, subchannel 0 of C_1 consists of slot 0 of every block and broadcasts pages 5 and 6.



(a) Page-set schedule (b) Page schedule

Channel C_0	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Channel C_1	5	11	7	9	6	12	8	10	5	13	7	9	6	11	8	10	5	12	7	9
Time Block	0				1				2				3				4			

(c) The broadcasting page sequence according to the page schedule

Subch# 0	5				6				5				6				5			
Subch# 1		11				12				13				11				12		
Subch# 2			7				8				7				8				7	
Subch# 3				9				10				9				10				9

Time →

(d) Four subchannels of channel C_1

Figure 4.2 The page-set and page schedule for K=2, β=4

Let us now check whether the page schedule in Figure 4.2(b) meets the requirement for continuous display. We assume a client starts to play a video at the beginning of time block *i*. Thus, pages 1 to 4 will be displayed in block *i*, pages 5 to 8 in block *i*+1, pages 9 to 12 in block *i*+2, and page 13 in block *i*+3. We can see this page schedule guarantees that each page has been received or will be received at the time it is needed. Take pages 7 to 10 for instance. At the beginning of slot 2 of block *i*+1, pages 7 and 8 either have been received or will be received from subchannel 2 of channel C_1, since both of them appear once every two blocks as shown in Figure 4.2(d). Thus, page

7 can be displayed on time. At the beginning of slot 3 of block *i*+1, page 8 has been received and can be displayed on time. Pages 9 and 10 will be received at slot 3 of either block *i* or block *i*+1, since both of them appear once every two blocks as shown in Figure 4.2(d). Thus, pages 9 and 10 can also be displayed on time at the beginning of slot 0 and 1 of block *i*+2, respectively.

According to the above description, we need to find a page-set schedule first, and then to map pages into page-sets in a strictly sequential fashion. We now formally define our HGWS problem as follows.

Definition: Given a video *V* of duration *D* seconds, block size β, and a set of *K* channels, the *Harmonic Group Window Scheduling* (*HGWS*) *Problem* is to find a partition of *V* into *n* page-sets, *PS*(1), *PS*(2), ..., *PS*(*n*), such that *PS*(*k*), where $0<k\leq n$, consists of the pages with a broadcasting period of *k* blocks, and to find a placement of the *n* page-sets to the $K\beta$ subchannels of the *K* channels. The placement should guarantee that for any viewer starting to play the video at the beginning of any block, each page will be received or has been received at the time slot when the viewer needs to consume the page. ■

Suppose s_1, s_2, ..., s_n subchannels are allocated to the above *n* page-sets, respectively. Since *i* pages from *PS*(*i*) can be mapped to a subchannel, the total number of pages in these *n* page-sets is $N=\sum_{i=1}^{n} is_i$. Each page has an equal size Db/N and each channel is divided into time slots of length $\delta = D/N$. Our goal is to maximize the total number of pages *N*, or equivalently to minimize D/N.

To schedule each page-set on some of the $K\beta$ subchannels, we need to guarantee that each page can be downloaded before or exactly when it is consumed so that the video can be displayed continuously. Since the pages of page-set 1 must be broadcast in every block and displayed immediately while downloading, we map the first β pages of a video to page-set 1 and allocate the β subchannels of C_0 to them so that they can be broadcast repeatedly in C_0 to guarantee continuous display. As for the pages in other page-sets, the following lemma gives a necessary condition to solve the HGWS problem. First we define a set of pages $P(k)$={ page i | $(k-1)\beta < i \le k\beta$ }, for $1 \le k \le n$.

Lemma 4.1: For any solution to the HGWS problem, each page in $P(k)$ must appear at least once on one of the K channels in every k consecutive blocks. If a client starts playing the movie at the beginning of block j, then the pages of $P(k)$ will be displayed in block $j+k-1$.

Proof: If a client starts to play and download a video from slot 0 of block j, page i will be displayed in slot ((i–1) mod β) of block $j+\lfloor (i-1)/\beta \rfloor$. Recall that pages are numbered 1, 2, ... For example, for β = 5, page 12 will be displayed in slot 1 of block j+2. Because clients may start from the beginning of any block, page i must appear at least once every $\lfloor (i-1)/\beta \rfloor + 1 = \lceil i/\beta \rceil$ blocks. If $(k-1)\beta < i \le k\beta$, then $\lceil i/\beta \rceil = k$, so all pages in $P(k)$ must appear at least once every k blocks and will be displayed in block $j+k-1$ if a client starts to view a movie at the beginning of block j. ■

According to the above lemma, we call $P(k)$ the ***ideal page-set*** k. We have the following theorem for upper bounds on the number of pages and page-sets.

Theorem 4.1 Given K channels, let integer n satisfy

$$\frac{1}{1}+\frac{1}{2}+\bullet\bullet\bullet+\frac{1}{n} \le K < \frac{1}{1}+\frac{1}{2}+\bullet\bullet\bullet+\frac{1}{n+1} \qquad (4.1.1)$$

Then the number of pages *N* cannot be more than $n\beta + (K - \sum_{i=1}^{n}\frac{1}{i})(n+1)\beta$ and the number of page-set (*N/β*) cannot be more than

$$n + (K - \sum_{i=1}^{n}\frac{1}{i})(n+1) \qquad (4.1.2)$$

Proof: The number of pages in *P(k)* is *β*, and according to Lemma 4.1, all pages in *P(k)* must be broadcast at least once in every *k* consecutive blocks (or once in every *kβ* consecutive slots). Thus, the pages in *P(k)* will consume at least *β*•(1/(*βk*)) = 1/*k* of the channel bandwidth. Since the total bandwidth cannot exceed *K*, we must have $\sum_{k=1}^{n}\frac{1}{k} \le K$. The maximum such *n* is given by Equation (4.1.1). Note that *K* channels together have *Kβ* subchannels. Of these, at least $\beta\sum_{i=1}^{n}\frac{1}{i}$ subchannels are allocated to page-sets *P*(1) to *P*(*n*). In other words, up to $(K - \sum_{i=1}^{n}\frac{1}{i})\beta$ subchannels may be still unallocated. This means that we can map at most $(K - \sum_{i=1}^{n}\frac{1}{i})\beta(n+1)$ pages to page-set *n*+1. Thus, *nβ*+$(K - \sum_{i=1}^{n}\frac{1}{i})\beta(n+1)$ is an upper bound on the number of pages. ■

From the above theorem, we can see that *P(i)* in HGWS is equivalent to segment S_i in the optimal harmonic windows scheduling problem, mentioned in Section 3.1, since *P(i)*, as well as S_i, consumes at least 1/*i* of the channel bandwidth. In HGWS, by dividing *P(i)* into many pages, the required bandwidth 1/*i* is separated into many small bandwidth portions so that we can use the residual bandwidth mentioned in Theorem 4.1 to pack a partial page-set.

Since a client needs to wait at most one block to start watching the video, and on average needs to wait for half a block, we have following corollary:

Corollary 4.1: Given a video with duration D seconds, $D/(n+(K-\sum_{i=1}^{n}\frac{1}{i})(n+1))$ is a lower bound on the maximum waiting time and $0.5D/(n+(K-\sum_{i=1}^{n}\frac{1}{i})(n+1))$ is a lower bound on the average waiting time. ■

According to the proof of Lemma 4.1, we need to guarantee that page *i* should be available at slot ((*i*–1) mod *β*) of block $j+\lfloor (i-1)/\beta \rfloor$ or before, if a client starts to play a movie at the beginning of block *j*. Since block *j* can be any block, page *i* should be mapped to subchannel *s*, where 0≤ *s* ≤ ((*i*–1) mod *β*), if it appears exactly once every $\lceil i/\beta \rceil$ blocks. Otherwise, its period should be less than $\lceil i/\beta \rceil$ blocks. Because a subchannel is allocated to only one page-set, not all the first pages (page (*k*-1)*β*+1) of all the ideal page-sets *P(k)* (*k*>0) can be mapped to subchannel 0 of some channel. For the same reason, not all the *i*-th pages (page (*k*–1)*β*+*i*) of all the ideal page-sets *P(k)* (*k*>0) can be mapped into subchannel *s* of some channel, where 0≤ *s*≤ (*i*–1). Thus, some pages in *P(k)* must be given a block period less than *k* to meet the continuity requirement. For example, pages 9 and 10 in Figure 4.2 (*β*=4) belong to the ideal page-set *P*(3) and need to appear at least once every three blocks, but actually they appear once in every two blocks. If we let page 9 be broadcast once every three blocks, page 9 should be mapped to subchannel 0 to guarantee that a client can reach or has received page 9 for playing after 3 time blocks from the moment the client started viewing and downloading the movie from a server. However, pages 5 and 6 of page-set 2 have been already mapped into subchannel 0. Therefore, page 9 should be broadcast with a period less than 3. Since each subchannel can broadcast two pages with a two block period, we promote both pages 9 and 10 to page-set 2.

Recall that *PS(k)* denotes the set of pages that are actually given block period *k*. Note that $PS(k) \subseteq P(k) \cup P(k+1)$. We call the pages of *P(k*+1) contained in *P(k*+1)∩*PS*(*k*) ***promoted*** **pages**.

Lemma 4.2 Page *i* ∈ *PS*(*k*)∩*P(k)*, where $k=\lceil i/\beta \rceil$, must be in subchannel *j* satisfying $0 \le j \le i-(k-1)\beta-1$ to guarantee a jitter free display. The promoted pages, i.e., those in *PS*(*k*)∩*P(k*+1*)*, can be put in any subchannels from 0 to β–1

Proof: The first part has been proved before. In the second part, each page in *PS*(*k*)∩*P*(*k*+1) will have been downloaded by the time when it is needed, no matter which subchannel it may be broadcast in. ■

To maximize the number of pages *N*, we must minimize the number of promoted pages from each *P(k*), since promoted pages consume more bandwidth than necessary. Let |*PS*(*i*)| and |*P*(*i*)| denote the number of pages in *PS*(*i*) and *P*(*i*), respectively. Further, let ***promoted*(*k*)=** |*PS*(*k*-1)∩*P(k)*| denote the number of pages actually promoted from *P(k*) and let ***subChs(i)*** denote the number of subchannels or the number of slots per block that the pages of page-set *i* occupy. The pages in *PS*(*i*) should include |*P*(*i*)|–*promoted*(*i*) pages from *P*(*i*) and *promoted*(*i*+1) pages promoted from *P(i*+1), i.e.,

$$|PS(i)| = \beta - promoted(i) + promoted(i+1) \tag{4.1.3}$$

Clearly *promoted*(1)=0. Since $\sum_{j=1}^{i-1}|P(j)| + promoted(i) = \sum_{z=1}^{i-1}|PS(z)|$ and $|P(i)|=\beta$, we clearly have

$$promoted(i) = \sum_{z=1}^{i-1} |PS(z)| - (i-1)\beta$$
$$promoted(1) = 0 \qquad (4.1.4)$$

We have the following necessary and sufficient condition to solve the HGWS problem by promoting some pages from each *P(k)*. Subchannel allocation is represented by $\delta_{i,j}$ such that $\delta_{i,j}$=1 (0≤ *j* < β) if subchannel *j* is allocated to page-set *i;* else $\delta_{i,j}$=0. Therefore, we have

$$subChs(i) = \sum_{z=0}^{\beta-1} \delta_{i,z} \qquad (4.1.5)$$

Let ***lBound$_\delta$(i)***, ***lBound*** of page-set *i*, denote the minimum number of pages that must be promoted from set *P(i)* to guarantee the continuous display of all the pages in page-set *i* under assignment δ.

Theorem 4.2: For any solution to the HGWS problem, given $\delta_{i,j}$ for 1≤ *i* ≤ *n* and 0≤ *j* ≤ β-1, we have

$$lBound_\delta(i) = \max\{j + 1 - i\sum_{z=0}^{j} \delta_{i,z} \mid 0 \le j < \beta\}$$

and

promoted(*i*)+ *i*• *subChs*(*i*) ≥ β.

Proof: We assume a client starts viewing and downloading pages from slot 0 of block *s*, so *P(i)* will be displayed in block *s* + *i* -1. By definition, the number of subchannels allocated to page-set *i*, between subchannel 0 and *j* inclusive, is $\sum_{z=0}^{j} \delta_{i,z}$.

We know that each subchannel $\mathscr{X}$ allocated to page-set *i* broadcasts *i* consecutive pages of page-set *i*, and these *i* consecutive pages, pages $\mathscr{X}_1$ to $\mathscr{X}_i$, are downloaded completely at the end of slot $\mathcal{X}$ of block *s*+*i*–1, as shown in Figure 4.3. Moreover, consecutive pages are mapped into page-set *i* in a strictly sequential fashion in the increasing order of their page numbers. Thus, at the end of any slot *j*, $0 \le j < \beta$, of block *s*+*i*–1, the number of downloaded consecutive pages of *P(i)*, counting from the first page of *P(i)*, is $promoted(i) + i\sum_{z=0}^{j}\delta_{i,z}$, and the number of displayed pages of *P(i)* is *j*+1. Therefore, to meet the continuity requirement of *P(i)* at the end of slot *j* of all blocks, the necessary and sufficient condition is $promoted(i) + i\sum_{z=0}^{j}\delta_{i,z} \ge j+1$, i.e., $promoted(i) \ge j+1-i\sum_{z=0}^{j}\delta_{i,z}$ and this condition should be satisfied for any *j*, $0 \le j < \beta$. Thus, *promoted*(*i*) must not be less than $\max\{j+1-i\sum_{z=0}^{j}\delta_{i,z} \mid 0 \le j < \beta\}$. This proves the first part of the theorem.

Letting $j=\beta$ in *promoted*(*i*) $\ge (j+1) - i\sum_{z=0}^{j}\delta_{i,z}$ and using (4.1.5), we get *promoted*(*i*)$\ge \beta - i\sum_{z=0}^{\beta-1}\delta_{i,z} = \beta - i \bullet subChs(i)$ which proves the second part. ■

Block#		Subchannel $\mathcal{X}$ or slot $\mathcal{X}$	
s	• • •	$\mathcal{X}_1$	• • •
s+1	• • •	$\mathcal{X}_2$	• • •
•	• • •	•	• • •
•	• • •	•	• • •
•	• • •	•	• • •
s+*i*–1	• • •	$\mathcal{X}_i$	• • •

Figure 4.3 Downloading pages from a subchannel

Since *promoted*(i)< β, *lBound*$_\delta$(i), a lower bound on *promoted*(i), is also always less than β. We now want to compute |*PS*(i)| for a given $\delta_{i,j}$ for 1≤ i≤ n and 0≤ j≤ β-1. According to Formula (4.1.4), *promoted*(i+1) can be calculated only after |*PS*(i)| is known, so we use *lBound*$_\delta$(i+1) instead of *promoted*(i+1) to calculate a lower bound on |*PS*(i)| from Formula (4.1.3). Summing both sides of (4.1.3) from 1 to i, we obtain:

$$\sum_{z=1}^{i} |PS(z)| = i\beta - promoted(1) + promoted(i+1)$$

$$= i\beta + promoted(i+1)$$

This can be rewritten as

$$|PS(i)| = i\beta - \sum_{z=1}^{i-1} |PS(z)| + promoted(i+1) \qquad (4.1.6.1)$$

$$\geq i\beta - \sum_{z=1}^{i-1} |PS(z)| + lBound_\delta(i+1)$$

Since each subchannel broadcast i consecutive pages of page-set i, the number of pages in *PS*(i) should be an integer multiple of i. Therefore, we have the following formula.

$$|PS(i)| = i\beta - i\left\lceil \frac{\sum_{z=1}^{i-1} |PS(z)| - lBound_\delta(i+1)}{i} \right\rceil \text{ where } i > 1 \text{ and } |PS(1)| = \beta \qquad (4.1.6)$$

In Formula (4.1.6), we use a ceiling function to guarantee that at least *lBound*$_\delta$(i+1) pages are promoted from page-set i + 1. We thus obtain |*P*(i)| and *promoted*(i) using the total number of pages in all the previous page-sets and the

$lBound_\delta(i)$ on the next page-set. For a solution to the HGWS problem, subchannel allocation, i.e., $\delta_{i,j}$ for $1\leq i\leq n$ and $0\leq j\leq \beta$-1, should satisfy

$$|PS(i)| = i * \text{subChs}(i) \quad \text{for } 1\leq i\leq n$$

where $|PS(i)|$ is obtained from Formula (4.1.6) and subChs(i) is from Formula (4.1.5).

Normally, it is not easy to get a right subchannel allocation satisfying the above condition directly. The method used in Section 4.2 is that we choose a rough subchannel allocation first, and then, for each page-set, we adjust the number of subchannels or slots to exactly meet the number of pages obtained from Formula (4.1.6) according to Theorem 4.2. If we succeed in the above adjustment of the number of subchannels for each page-set, we get the final solution.

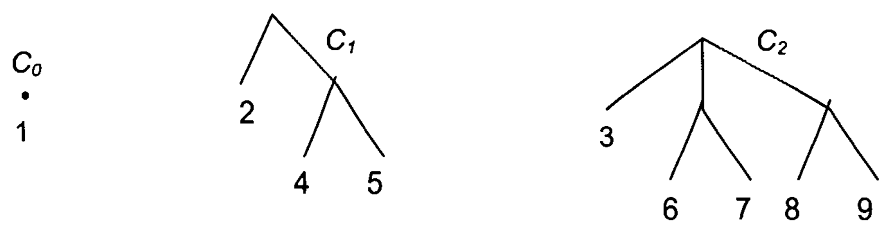
4.2 HPB Scheme

In this section, we will introduce our detailed solution to the HGWS problem, i.e., the HPB scheme. Given a set of page-sets satisfying the conditions given in Section 4.1, how can we assign all the page-sets into βK subchannels so that we can pack as many pages as possible into K channels with block size β?

As mentioned in Section 4.1, to maximize the total number of packed pages N, we should try to minimize the number of promoted pages. Therefore, we should put as many pages of the ideal page-set $P(i)$ as possible in the actual page-set $PS(i)$ so that, for all i = 1, 2, ..., page-set $PS(i)$ will occupy a bandwidth as close as possible to $1/i$ of the channel bandwidth, which implies roughly β/i subchannels should be allocated to page-set $PS(i)$. We know that each segment S_i in the schedule of RFS or GFDPB (m=1) consumes no less than and close to $1/i$ of the channel bandwidth. So they give us something that approximates our goal.

Given a schedule from either RFS or GFDPB, we can get a segment broadcasting sequence for each channel by repeating the corresponding channel schedule as mentioned in Section 3.1. Suppose we pick a suitable β and the first one block (or β slots) of the segment broadcasting sequence of each channel. If we now regard each segment i as page-set i and each slot j ($0 \leq j < \beta$) as subchannel j, we will get an initial version of the ***rough page-set schedule*** in which each page-set is assigned to roughly β / i subchannels. We call the corresponding schedule from RFS or GFDPB (m=1) the initial version of the ***rough block schedule,*** where each segment label should be interpreted as a page-set label. For example, let us consider the case where K=3 and β=18. The schedule from RFS or GFDPB (m=1) is shown in Figure 4.4(a). Figure 4.4(b) is the segment broadcasting sequence of schedule (a) in the first 18 slots of the three channels. If we regard slot i ($0 \leq i \leq 17$) of each channel in (b) as subchannel i of the corresponding channel, then (b) is the initial version of the rough page-set schedule for K=3 and β=18, and (a) is the initial version of the rough block schedule. Later, we will talk about how to improve this initial version of the rough block schedule. First, we will try to increase the number of packed page-sets and get a final version of the rough block schedule. Second, we will modify the final version of the rough page-set schedule using Theorem 4.2 to get the actual page-set schedule which exactly meets the requirement of continuous display.

According to Theorem 4.2, we have $i \bullet subChs(i) \geq \beta - promoted(i)$, for any page-set i. It means the number of subchannels allocated to page-set i is sufficient to hold all the (β - $promoted(i)$) pages of the ideal page-set $P(i)$, excluding the promoted pages. Note that each subchannel can accommodate i pages of $PS(i)$. According to Lemma 4.2, promoted pages can be put into any surplus subchannels of page-set j satisfying $j \bullet subChs(j) > |PS(j)|$.



(a) The initial version of the rough block schedule for K=3

Subchannel# or Slot #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
C_0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
C_1	2	4	2	5	2	4	2	5	2	4	2	5	2	4	2	5	2	4
C_2	3	6	8	3	7	9	3	6	8	3	7	9	3	6	8	3	7	9

(b) The initial version of the rough page-set schedule for K=3 and β=18

Figure 4.4 Rough block schedule and rough page-set schedule for K=3

If page-set j has a period p [1] less than j in a rough block schedule (tree), then it may occupy more subchannels in the corresponding rough page-set schedule than necessary to support all the pages in $PS(j)$, i.e., the interval p between any two consecutive appearances of the (pages of) page-set j in the corresponding rough page-set schedule is less than j time slots in a block. We say in this case that ***pageset j has free space*** or ***free bandwidth*** ($1/p - 1/j$). There are many page-sets of such kind in an initial version of a rough block schedule.

We call a page-set which is assigned to only one leaf in a rough block schedule a ***perfect page-set***, otherwise we call the page-set a ***non-perfect page-set***. Since we only use a perfect schedule as our initial version of a rough block schedule, all the page-sets in the initial version of the rough block schedule are perfect page-sets. Later, we show that we can add some non-perfect page-sets to the initial version of a rough block schedule to increase the number of packed page-sets.

[1] All pages in page-set j have the same period, j block, in the actual page schedule.

Lemma 4.3 Given a schedule of HPB, the *lBound*$_\delta$*(i)* of a perfect page-set *i*, computed from a rough block schedule, is equal to its start slot s_0^i, the smallest subchannel number allocated to this page-set.

Proof: We use the same assumptions as in Theorem 4.2 and let $lBound_\delta(i,j) = (j+1) - i\sum_{z=0}^{j}\delta_{i,z}$ for some *j*. Since $\delta_{i,z} = 0$ for $z < s_0^i$, we have max{$lBound_\delta(i,j)$ | $0 \le j < s_0^i$ }= s_0^i. Consequently, we have *promoted*(*i*) $\ge s_0^i$. Suppose a client starts viewing and downloading pages from slot 0 in block *s*. The pages of *P(i)* are displayed in block *s* + *i* -1. Note that the interval *p* between any two consecutive appearances of the (pages of) perfect page-set *i* in the corresponding rough page-set schedule is not more than *i* time slots in a block, i.e., *p* ≤ *i*. Moreover, each subchannel $\mathscr{X}$ allocated to page-set *i* broadcasts *i* consecutive pages of page-set *i* and these *i* consecutive pages can be downloaded completely by the end of slot $\mathscr{X}$ of block *s*+*i*–1. Therefore, the number of downloaded consecutive pages of *P(i)*, counting from the first page of *P(i)*, increases by *i* every *p* slots in block *s* + *i* –1 after slot s_0^i. Therefore, if *promoted*(*i*) $\ge s_0^i$, the continuous display of the perfect page-set *i* is guaranteed. ■

From the proof of Lemma 4.3, we can guarantee that each page-set *i* with a period equal to *i* in a rough block schedule can be displayed on time, if the number of promoted pages is equal to its start slot number. As we stated above, there are many page-sets with free space that is more than enough to support promoted pages in the initial version of a rough block schedule. We can assign the free bandwidth of these page-sets to some new page-sets in a rough block schedule without affecting the continuity of the former page-sets. In this way, we can modify the initial version of a rough block schedule to increase the number of packed page-sets, within the upper bound given in Theorem 4.1. We call a page-set in the initial version of a rough block schedule ***initial page-set***, otherwise ***extended page-set***. For example, for K=3, the initial version of the rough block schedule obtained directly from RFS or GDPB (m=1) has 9 page-sets, and we can insert page-set 10 and a partial page-set 11 into the initial page-sets having free bandwidth. Here, page-sets 1 to 9 are the initial page-sets and page-sets 10 and 11 are the extended page-sets.

Next, we define *extra_slot*(i, j) as a measure of which subchannel j, originally allocated to an initial page-set i having free space, can be reallocated to an extended page-set in a rough page-set schedule.

extra_slot(i, j) indicates the number of consecutive pages already downloaded in the buffer space at the beginning of slot j of ***the i-th block of display time***, i.e., the time block during which a client plays the ideal page-set $P(i)$ of a video, in the worst case. The consecutive pages are counted from page $(i-1)\beta+j+1$, where β is the block size. The worst case for the i-th block of display time is when each slot in the block, whose corresponding subchannel is allocated to page-set i, is for broadcasting the first page of the i consecutive pages mapped to the corresponding subchannel of the slot. Page $(K-1)\beta+j+1$ is the page just needed to be consumed at the beginning of slot j of the i-th block of display time. In other words, *extra_slot*(i, j) is equal to the number of downloaded consecutive pages of $P(i)$, counting from the first page of $P(i)$, minus the number of pages of $P(i)$ having consumed at the beginning of slot j of the i-th block of display time for the worst case scenario mentioned above.

For example, in Figure 4.2, we have $P(3)$={page i | $9 \le i \le 12$}, $PS(2)$={page i | $5 \le i \le 10$}, $PS(3)$={page i | $11 \le i \le 13$}, and *start_slot*(3)=1, so we have *promoted*(3) = $|PS(2) \cap P(3)|$=2. Take the 3rd block of display time for instance. The pages of $P(3)$, i.e., pages 9 to 12, will be displayed in the block. Since pages 9 and 10 are promoted from

P(3) to *PS*(2), they will be downloaded in the 1st or the 2nd block of display time. There are three pages, i.e., pages 11, 12, and 13, in *PS*(3), each of which will be downloaded in the 1st, 2nd, or 3rd block of display time. Thus, the 3rd block of display time has three different cases related to the downloading of *PS*(3): the block broadcasting page 11, the block broadcasting page 12, and the block broadcasting page 13, as shown in Figure 4.2(c). At the beginning of slot 1 of the 3rd block of display time, when page 10 is needed for display, if the 3rd block is the block broadcasting page 11, then the number of the downloaded consecutive pages of *P*(3) counting from page 10 in the buffer space is 1 (page 10). If the 3rd block is the block broadcasting page 12, then only two such pages (pages 10 and 11) are in the buffer. If the 3rd block is the block broadcasting page 13, then only three such pages (pages 10, 11 and 12) are in the buffer. Thus, the worst case for the 3rd block in Figure 4.2(c) is the block broadcasting page 11, i.e., the first page of the three consecutive pages mapped to subchannel 1, and *extra_slot*(3, 1)=1.

If subchannels j_0 and j_1 are two adjacent subchannels allocated to page-set i and j_0+*extra_slot*(i, j_0) ≥ j_1, then we can reallocate subchannel j_0 to an extended page-set to replace page-set i in a rough page-set schedule without changing the continuity of the display of page-set i. The reason is that at the beginning of slot j_0, *extra_slot*(i, j_0) can support continuous display up to the beginning of slot j_1 in the worst case for the i-th block of display time. If page-set i is a perfect page-set and its period is p slots in a rough block schedule, we have $j_0 + p = j_1$; moreover, if *extra_slot*(i, j_0) ≥ p, we have j_0+*extra_slot*(i, j_0) ≥ j_1.

In Figure 4.4, we present an insertion algorithm for assigning the free bandwidth of a perfect initial page-set to an extended page-set. It splits the leaf with the initial page-set label in a rough block schedule into several child leaves, and then assigns some of those child leaves to the extended page-set. In the algorithm, a leaf with a label of an initial page-set *i* in a rough block schedule is split into *split_num* child leaves, so the original subchannels allocated to page-set *i*, or the corresponding slots in each block, are divided into groups of *split_num*, each of which consists of *split_num* original consecutive subchannels or slots for page-set *i*. Thus, in the first group, the first original subchannel allocated to page-set *i*, or the corresponding slot in each block, corresponds to the first child leaf, and the second original subchannel or slot for page-set *i* corresponds to the second child leaf, ..., and the (*split_num*)-th original subchannel or slot for page-set *i* corresponds to the last child leaf. The above matching is repeated again in the next group of *split_num* original consecutive subchannels or slots for page-set *i*: the (*split_num*+1)-th original subchannel or slot for page-set *i* corresponds to the first child leaf, and so on.

```
Algorithm: Assign the free bandwidth of a perfect initial page-set to an extended
        page-set in a rough block schedule
Input: Trees of a rough block schedule, an extended page-set a, an initial page-set i
        with start_slot(i)=s, promoted(i)=r, and a period p
Ouput: The rough block schedule with the extended page-set a inserted into the
        subtree of the original page-set label i
  1) Split the leaf with page-set label i into split_num=i/GCD(i, i-p) child leaves and
     label the child leaves 0, 1, 2, ..., split_num-1, where GCD means the
     greatest common divisor. The number of leaves given to extended page-set a
     is new_num=(i-p)/GCD(i, i-p).
  2) Initialize extra_slot=(r-s) for the first child leaf and set j=1.
  3) If extra_slot ≥ p, assign extended page-set a to the j-th child leaf,
             increment j by 1, and update extra_slot=extra_slot – p
     else
             assign page-set i to the j-th child leaf, increment j by 1 and update
             extra_sot = extra_slot+i-p
  4) If j < split_num, go to step 3, else output the trees.
```

Figure 4.5 The insertion algorithm for HPB

Each group of *split_num* original consecutive slots for page-set *i* in a block repeats the same matching pattern. Therefore, in Figure 4.4, when we calculate the value of *extra_slot* of a child leaf, we regard the child leaf as its corresponding slot in the first group of *split_num* original consecutive slots of page-set *i* in the *i*-th block of display time. Because we always keep the value of *extra_slot* of the next child leaf in step 3 of Figure 4.4, and we already know the corresponding slot and block of each leaf, we only use the variable *extra_slot* in the algorithm without indicating the corresponding parameters. If a child leaf corresponds to slot *k* and is assigned to page-set *i*, then all the *i* pages broadcast in the corresponding subchannel *k* must have been downloaded at the end of slot *k* of the *i*-th block of display time. At the beginning of the next slot, slot *k+p*, whose corresponding subchannel is allocated to page-set *i*, the value of *extra_slot* is increased by *i* - *p* from the value at the beginning of slot *k*, as shown in step 3 of Figure 4.4. The reason is that during the period between the beginning of slot *k* and the beginning of slot *k+p* in the *i*-th block of display time in the worst case, the number of downloaded consecutive pages of *P(i)*, counting from the first page of *P(i)*, increases by *i* and *p* pages are consumed.

Theorem 4.3 proves that we can pick exactly *new_num* subchannels from each group of *split_num* original consecutive subchannels of page-set *i* to be assigned to extended page-set *a* (in Figure 4.4) without changing the continuity of the display of page-set *i*, i.e., without changing the value of *promoted*(*i*).

Theorem 4.3: The algorithm in Figure 4.5 can guarantee that a total number of *new_num* leaves for page-set *a* can be added, and page-set *i* occupies exactly 1/*i* of the channel bandwidth in the output trees. Page-set *a* gets (1/*p* – 1/*i*) of the channel bandwidth after being inserted.

Proof: We assume that we can assign y child leaves to the extended page-set *a*. From step 3 of Figure 4.5 we see that every child leaf assigned to page-set *i* contributes (*i* - *p*) slots to *extra_slot*, so the total contribution to the value of *extra_slot* through the (*split_num*- $\mathscr{y}$) child leaves assigned to page-set *i* is (*split_num* - $\mathscr{y}$)*(*i* - *p*). Also from step 3, we see that each child leaf assigned to extended page-set *a* consumes *p* slots from *extra_slot*, so we need to consume a total of $\mathscr{y}p$ slots from *extra_slot* for the $\mathscr{y}$ child leaves. Thus, for guaranteeing that each group of *split_num* original consecutive slots for page-set *i* in a block repeats the same insertion pattern, we should have

$$(split\_num - \mathscr{y})*(i - p) = \mathscr{y}p$$

$$\mathscr{y} = split\_num * (i - p)/i = (i - p)/\mathrm{GCD}(i, i - p) = new\_num$$

Therefore, we can assign exactly *new_num* child leaves to extended page-set *a* in the algorithm.

Since the leaf for the initial page-set *i* is split into *split_num* child leaves in which *new_num* leaves are given to the extended page-set *a*, (*split_num*-*new_num*)=*p*/GCD(*i*, *i*-*p*) child leaves are assigned to page-set *i*. The bandwidth of each child leaf is 1/(*split_num*•*p*), so the total bandwidth of page-set *i* in the output trees is (1/(*split_num*•*p*))• *p*/GCD(*i*, *i*-*p*)=1/*i*. Thus, the residual bandwidth (1/*p* – 1/*i*) is given to page-set *a*. Assigning a leaf to the extended page-set if *extra_slot* ≥ *p* in step 3 of the algorithm guarantees the continuous display of page-set *i* after the assignment. ■

Consider, for instance, the case of inserting extended page-set 11 into the free space of the initial page-set 8 with a period of six slots in a rough block schedule given *promoted*(8)=7 and *start_slot*(8)=2. In step 1 of Figure 4.5, we get GCD(8, (8-6))=2, *split_num*=8/2=4, and *new_num*=(8-6)/2=1. When we calculate the value of *extra_slot* of a child leaf, we regard the child leaf as its corresponding slot in the first group of four original consecutive slots, slots 2, 8, 14, and 20, for page-set *i* in the *i*-th block of display time. In step 2, we initialize *extra_slot*=7-2=5 for the first child leaf. In step 3, we increase *extra_slot* by 8-6=2 every time for assigning page-set 8 to a child leaf and decrease *extra_slot* by 6 every time for assigning page-set 11 to a child leaf as shown in Figure 4.6. For the last, or the rightmost, child leaf, we have *extra_slot*=3, so we assign page-set 8 to it and update *extra_slot*=*extra_slot*+2=5, which is exactly the same as the value of *extra_slot* of the first child leaf. Therefore, each group of four original consecutive slots of the initial page-set 8 in each block repeats the same insertion pattern as shown in Figure 4.6.

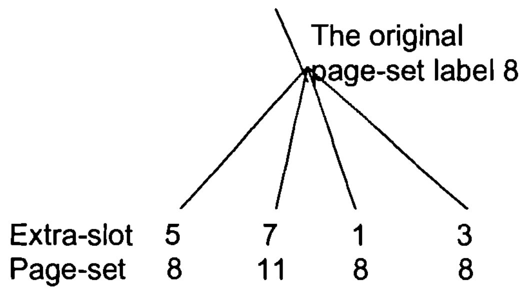


Figure 4.6 An example for insertion algorithm

To minimize the number of promoted pages, we try to allocate 1/*a* of a channel bandwidth to extended page-set *a*. To do so, we need to choose several initial page-sets that have free space and the sum of whose free bandwidths is greater than or equal to 1/*a* to provide page-set *a* with sufficient free bandwidth according to the algorithm shown in Figure 4.5. Thus, most of the extended page-sets are non-perfect page-sets, each of which is assigned to more than one leaf in a rough block schedule after being inserted completely.

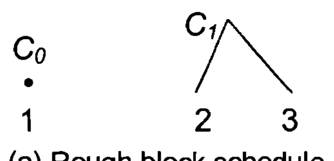
We denote each leaf of a non-perfect page-set *i* in a rough block schedule as (s_j^i, p_j^i) in which s_j^i represents the start slot of the *j*-th leaf of page-set *i*, and p_j^i represents the corresponding period. We assume that a non-perfect page-set *i* has α+1 leaves, (s_0^i, p_0^i), (s_1^i, p_1^i), $\cdots$, (s_α^i, p_α^i), in the increasing order of s_j^i ($0 \le j \le \alpha$). We call the *j*-th leaf of page-set *i the **subpage-set j of page-set i***, so the above page-set *i* has

α+1 subpage-sets. Since each leaf(s_j^i, p_j^i) represents a sequence of time slots [s_j^i, $s_j^i + p_j^i$, $s_j^i + 2p_j^i$, $s_j^i + 3p_j^i$, • • •] inside a block, beginning at slots_j^i, and repeating with a period of p_j^i slots, then we have the following formula. If page-set *i* is a perfect page-set, then α=0.

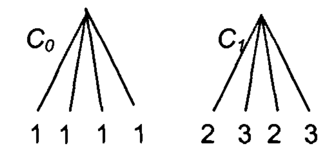
$$subChs(i) = \sum_{j=0}^{\alpha}(1 + \left\lfloor \frac{\beta - 1 - s_j^i}{p_j^i} \right\rfloor) \quad \text{where } \beta > s_\alpha^i \qquad (4.2.1)$$

Obviously, the number of the pages mapped into *subChs(i)* subchannels is equal to *i* times *subChs(i)*.

Now, we can talk about how to get the page-set schedule as shown in Figure 4.2. For *K*=2 and *β*=4, an initial version of the rough block schedule from RFS and the corresponding rough page-set schedule are shown in Figure 4.7 (a) and (b), respectively. First, we check the *lBound*$_\delta$*(i)* of each page-set according to Lemma 4.3: *lBound*$_\delta$(1)=0, *lBound*$_\delta$(2)=0, and *lBound*$_\delta$(3)=1. Second, we calculate the number of promoted pages and the number of pages of each page-set except the last page-set according to Formulae (4.1.4) and (4.1.6): *promoted*(1)=0, |*PS*(1)|=4, *promoted*(2)=4-4=0, and |*PS*(2)|= $\left\lceil \frac{4-0+1}{2} \right\rceil \times 2$ =6. For the last page-set (page-set 3), *lBound*$_\delta$(3) already guarantees its continuous display in the 3rd block of display time according to Lemma 4.3. Adding additional subchannels to it or giving some last subchannels of it to other page-sets doesn't affect its continuity. Therefore, the number of pages of the last page-set is very flexible. Third, we calculate the number of slots or subchannels of each page-set according to Formula (4.2.1) with α=0: *subChs(*1)=4, *subChs(*2)=2, and *subChs(*3)=2. Fourth, we adjust the number of subchannels of each page-set to fit the number of pages of the same page-set except the last page-set. Since each subchannel allocated to page-set *i* broadcasts *i* consecutive pages of page-set *i*, |*PS*(*i*)|/*i* is the number of the subchannels needed to support the number of the pages of page-set *i*. By comparing |*PS*(*i*)|/*i* with *subChs*(*i*), where 1 ≤ *i* ≤2, we find page-set 2 needs one more subchannel to support its pages. Therefore, we give the last subchannel of page-set 3, i.e., subchannel 3 in the rough page-set schedule (see Figure 4.7(b)), to page-set 2 and get the actual page-set schedule as shown in Figure 4.2(a).



(a) Rough block schedule



(b) Rough page-set schedule

Figure 4.7 HPB example for *K*=2, *β*=4

According to Theorem 4.2, we have the following lemma for the *lBound* of a non-perfect page-set.

Lemma 4.4: Assume that an extended page-set *a* has α+1 leaves after being inserted in a rough block schedule (trees) and the leaves, (s_0^a, p_0^a), (s_1^a, p_1^a), $\cdots$, (s_α^a, p_α^a), are in the increasing order of start slot s_j^a (0 ≤ *j* ≤ α).

Let $lBound_\delta(a,j) = (j+1) - a\sum_{z=0}^{j}\delta_{a,z} = (j+1) - a\sum_{z=0}^{y}(1+\left\lfloor\frac{j-s_z^a}{p_z^a}\right\rfloor)$, where 0 ≤ *y* ≤ α, *j* $\geq s_y^a$, and *y* = α or *j* $< s_{y+1}^a$, then $lBound_\delta(a) = \max\{lBound_\delta(a,j) \mid 0 \le j < \beta\}$.

Proof: For any leaf (s_i^a, p_i^a) of page-set *a*, the number of the subchannels corresponding to the leaf (subpage-set) between subchannel 0 and *j*, inclusive, is $1+\left\lfloor\frac{j-s_i^a}{p_i^a}\right\rfloor$ if *j* $\geq s_i^a$; else it is zero. The rest is similar to the proof of Theorem 4.2 after summing up over all the leaves (subpage-sets) of page-set *a*,. ■

Now, we present our HPB. Figure 4.8 is a flow chart of the algorithm. The main data structure in the algorithm is a page-set list, *pagesetlist*, where each object represents a leaf with a page-set label in a rough block schedule forest, so non-perfect page-sets have more than one page-set object. To meet the continuity requirement, first, we use Lemma 4.3 or Lemma 4.4 to calculate the values of *lBound* for each page-set, and then use Formula (4.1.6) to calculate the number of pages (indicated as *nPages* here) for each page-set. According to the algorithm in Figure 4.5, we insert extended page-sets up to the given page-set number, i.e., *max_pageset* in Figure 4.9, into the free space of initial page-sets having free bandwidth in the rough block schedule to get the final version of the rough block schedule.

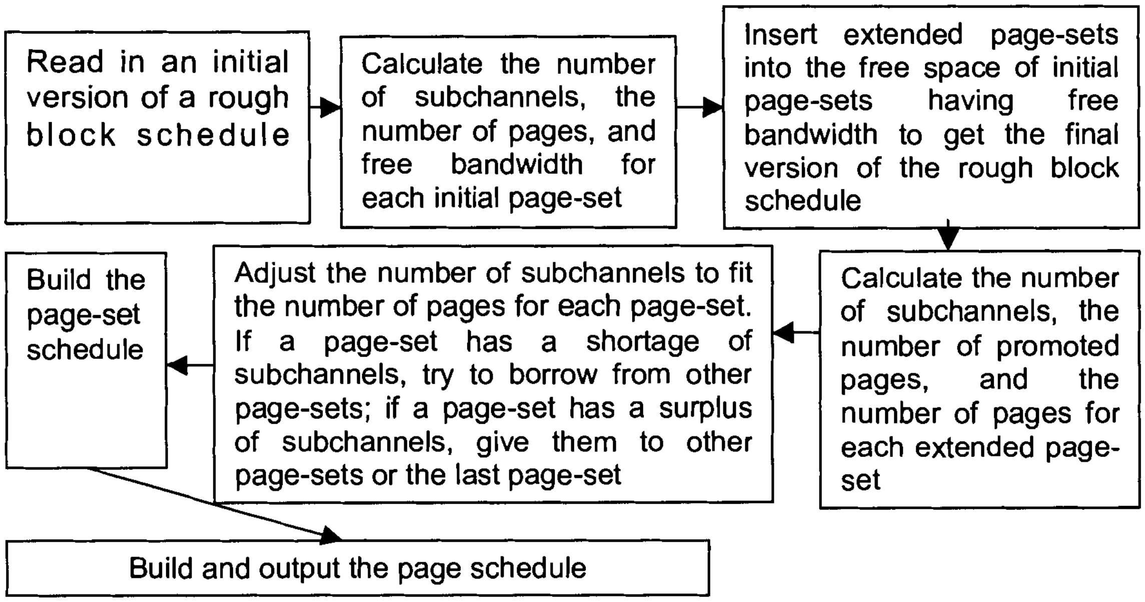

Read in an initial version of a rough block schedule
Calculate the number of subchannels, the number of pages, and free bandwidth for each initial page-set
Insert extended page-sets into the free space of initial page-sets having free bandwidth to get the final version of the rough block schedule
Calculate the number of subchannels, the number of promoted pages, and the number of pages for each extended page-set
Adjust the number of subchannels to fit the number of pages for each page-set. If a page-set has a shortage of subchannels, try to borrow from other page-sets; if a page-set has a surplus of subchannels, give them to other page-sets or the last page-set
Build the page-set schedule
Build and output the page schedule

Figure 4.8 Flowchart of HPB

Now, we talk about how to transfer subchannels from page-sets to page-sets to meet the continuous display requirement after getting the final version of the rough block schedule. Suppose that we have *subChs*(*i*) = $\mathcal{X}$ and |*PS*(*i*)|/*i* = $\mathcal{Y}$, and page-set *i* is assigned to subchannels i_1, i_2, ... , $i_{\mathcal{X}}$ (in the increasing order of subchannel numbers), each from one of the *K* given channels. If $\mathcal{X} > \mathcal{Y}$, page-set *i* has a surplus of subchannels.

According to Theorem 4.2 and Formula (4.1.6), *lBound*$_{\delta}$*(i)* used in calculating $|PS(i-1)|$ guarantees the continuous display of page-set *i* from slots 0 to $i_{\mathcal{Y}}$ in the *i*-th block of display time. After slot $i_{\mathcal{Y}}$ of the *i*-th block of display time, all the pages of page-set *i* have been downloaded in a client's buffer space. Thus, the continuous display of page-set *i* is still be guaranteed if we give subchannels $i_{\mathcal{Y}+1}$ to $i_{\mathcal{X}}$ to other page-sets that have a shortage of subchannels. If $\mathcal{X} < \mathcal{Y}$, page-set *i* has a shortage of subchannels. According to theorem 4.2, those pages that cannot be accommodated on the $\mathcal{X}$ subchannels are the promoted pages from page-set *i*+1. We can put the promoted pages into any subchannels borrowed from page-sets that have surplus subchannels and don't worry about their display continuity. For the last page-set (the last extended page-set), we also calculate its *lBound* according to lemma 4.4 so that adding additional subchannels to it, or giving some last subchannels of it to other page-sets, doesn't affect its continuous display. Thus, the number of pages of the last page-set is very flexible. If a page-set has a surplus of subchannels and no other page-sets need them, we give them to the last page-set. If a page-set has a shortage of subchannels and no other page-sets have surplus subchannels, then we borrow subchannels from the last page-set as indicated in step 9 of Figure 4.9.

List *pagesetlist* is a list of objects of a page-set class that has the following private member variables: *channel, start_slot, period, pageset, nPages, subChs, lBound, promoted, more, givelist,* and *borrowlist.* The first four variables indicate the place of a leaf with page-set label, *pageset,* in the rough block schedule tree. The next four variables are used to check for a sufficient condition for continuous display of this page-set. The last three variables, where *more* = *subChs* - *nPages*/*pageset,* are used to match the number of subchannels with the number of pages as described in Figure 4.8. We sort *pagesetlist* in increasing order of (*pageset, start_slot, channel*) so that all the leaves assigned to the same non-perfect page-set are put together in the increasing order of *start_slots*. For a non-perfect page-set, its parameters, *nPages, subChs, lBound, promoted, more, givelist,* and *borrowlist*, are stored in the first subpage-set.

Figure 4.9 shows the detailed algorithm of HPB. In step 1, we read in an initial version of a rough block schedule generated by GFDPB (*m*=1) or RFS, and build the *pagesetlist*, and then in step 3, we calculate parameters of each initial page-set: *lBound, promoted, nPages*, and *subChs.* In step 4, we build a list, *freelist*, which contains all the page-sets having free spaces and their corresponding free bandwidths. In step 5, for each extended page-set *i,* we choose a set of initial page-sets from the *freelist* to form a *freeD* list where the sum of all the free bandwidths exceeds and is close to 1/*i*, and then insert the extended page-set *i* into the free space of each of the page-sets in the *freeD* list according to the algorithm in Figure 4.5. Step 5c is a heuristic for reducing the waste of free bandwidth while adding an extended page-set. In step 6, we calculate *lBound, promoted, nPages*, and *subChs* for all the extended page-sets. In step 7, we calculate the value of *more* = *subChs* - *nPages*/*pageset* for each page-set. If a page-set lacks subchannels, i.e., *more*<0, we put it in the *lesslist;* if a page-set has a surplus of subchannels, i.e., *more*>0, we put it in the *morelist.* From steps 8 to 10, we adjust the number of subchannels, *subChs*, to fit the number of pages, *nPages*, for each page-set including all the initial and extended page-sets through the *morelist* and the *lesslist.* If a page-set lacks subchannels, then we borrow some subchannels from other page-sets that have a surplus of subchannels, or from the last extended page-set, and record the number of the subchannels borrowed and the page-set from which the subchannels have been borrowed in the *borrowlist* of the corresponding page-set. If a page-set has too many subchannels, then we give the surplus subchannels to other page-sets which lack subchannels, or give them to the last extended page-set, and then record the number of the subchannels and the page-set to which the subchannels have been given in the *givelist* of the corresponding page-set.

Now, we get the final version of the rough block schedule. In step 13a, we build ***the final version of the rough page-set schedule*** according to ***the final version of the rough block schedule*** indicated by *pagesetlist*. In *pagesetlist,* each object represents a leaf with a page-set label in the final version of the rough block schedule forest through four private member variables: *channel, start_slot, period*, and *pageset.* Then, in step 13b, we modify the above final version of the rough page-set schedule to make each page-set meet the continuous display requirement by giving their last several subchannels or slots to other page-sets in *givelist* and get the actual page-set schedule. Finally, we build the page schedule in step 14 by mapping pages into each page-set in a strictly sequential fashion. If there is no sufficient free bandwidth for adding the given number of extended page-sets, or if we fail in matching the number of subchannels with the number of pages for some page-sets, an error message will be generated and the program terminates.

The output of HPB is *pagelist* where each object represents a leaf with a page label in the actual page schedule forest. Each page object is indicated by four private variables. The first one is the channel number and the second is the start slot number. The third is the broadcasting period of the page (in slots), and the fourth is the page number. Note that the final page schedule is an actual broadcasting schedule like the GFDPB schedule and is ready for being put into our server multiplexing scheme in Section 3.7 for actual broadcasting. A time slot is the only time unit used in the page schedule. No time blocks and page-sets are used in *pagelist.*

```
Algorithm: HPB(K, β, max_pageset)
Input: Number of channels K, block size β, and max_pageset, which is the last
    page-set number you want to insert.
Output: Total number of pages and a page schedule, pagelist.
1) Read an initial version of a rough block schedule in the format (channel, star_slot,
   period, pageset) per line, build the pagesetlist, and count the number of initial
   page-sets, pageset_num.
2) Sort the pagesetlist in the increasing order of (pageset, start_slot, channel).
3) For each initial page-set i except the last, calculate subChs according to Formula
   (4.2.1) with $\alpha=0$, promoted according to Formula (4.1.4), lBound according to
   Lemma 4.3, nPages according to Formula 4.1.6, and more = subChs - nPages/i.
   For the last initial page-set, just calculate the value of promoted, since we don't
   know the lBound value of its next pageset, the first extended page-set, at this
   point.
4) For each initial page-set i with period p, if $i > p$, then this page-set has free
   bandwidth with period $freeP=i \cdot p/(i-p)$, whose corresponding bandwidth is
   $1/freeP$; put (freeP, i) into freelist in the increasing order of freeP.
5) For each extended page-set i from page-set pageset_num+1 to page-set
   max_pageset, do the following:
    5a) Create a null list freeD.
    5b) If the free bandwidth of the first page-set in freelist is not less than 1/i,
        remove it from the head of freelist, add it to freeD list, and then go to 5f).
    5c) If we can find an integer n such that the sum of the first n page-sets' free
        bandwidth from freelist is not larger than 1/i and the sum of the first n+1 free
        bandwidth is larger than 1/i, then do the following; else go to 5d) directly.
        Remove the first n page-sets from freelist and add them into freeD list. If the
        sum of all the free bandwidth in freeD is less than 1/i, go though the rest of
        freelist to pick the last free bandwidth for freeD list, which makes the sum of all
        the free bandwidths in freeD list closer to 1/i than any other free bandwidth in
        freelist and larger than 1/i, and then remove the corresponding page-set from
        freelist. Go to 5f).
    5d) If the sum of all the page-sets' free bandwidth in freelist is less than 1/i and
        i < max_pageset, then output an error message "Error: max_pageset should
        be changed to i" and terminate the program.
    5e) If i = max_pageset, then move all the remaining free bandwidths from
        freelist to freeD list.
    5f) For each page-set j in freeD list, do the following: Find the initial page-set j
        in pagesetlist, remove it, assign it to pre_PS, and then insert page-set i into the
        free space of page-set j according to Figure 4.5. Copy nPages and promoted
        of pre_PS to the first subpage-set of page-set j after insertion and calculate the
        new values of subChs and more for page-set j.
    5g) Calculate lBound of the just inserted page-set i according to Lemma 4.4.
```

6) Calculate *promoted, nPages*, *subChs*, and *more* for page-set *pageset_num* and all the extended page-sets except page-set *max_pageset*, whose *nPages* is flexible.

7) For each page-set *i*, if *more*>0, then put (*i, more*) into *morelist* in the increasing order of *more*; if *more*<0, then put (*i, -more*) into *lesslist* in the increasing order of *–more*.

8) For each page-set *i* in *lesslist*, whose *-more* is assigned to φ, do the following.

   8a) If *morelist* is empty, go to step 9.

   8b) Go through *morelist* to find the page-set, page-set *j*, whose *more* $\geq \varphi$. Remove page-set *j* from *morelist* and page-set *i* from *lesslist*. If *more* of page-set *j* is larger than φ, deduct φ from *more*, then put (*j, more-*φ) into *morelist* in order. Every pair of actions of borrowing subchannels and giving subchannels is recorded in the corresponding page-set's *borrowlist* and *givelist.*

   8c) If we can't find page-set *j* in 8b), then do as follows. Fetch and remove the page-set from the head of *morelist* and deduct the corresponding *more* value from φ and update φ=φ-*more*. Repeat the above operation until $\varphi \leq$ *more* value of a just fetched page-set, or *morelist* is empty. If *more* value of the just fetched page-set is larger than the updated φ, then put the remainder *more-*φ back to *morelist* in order. If we use up all the surplus subchannels in *morelist* and the updated φ is still larger than 0, then put the updated φ and page-set *i* back in the head of *lesslist*, then go to step 9. Also, every pair of actions of borrowing subchannels and giving subcahnnels should be recorded in the corresponding page-sets' *borrowlist* or *givelist.*

9) If *lesslist* is still not empty, then borrow subchannels from the last page-set (*max_pageset*). If all the subchannels of page-set *max_pageset* are used up and *lesslist* is still not empty, then output error message "Failure in matching *less* and *more*, try to decrease *max_pageset*!!!" and terminate the program.

10) If *morelist* is still not empty, then give all the surplus subchannels in *morelist* to the last page-set, page-set *max_pageset.*

11) Calculate the last page-set's *subChs* according to its subpage-sets, *borrowlist* and *givelist*; then calculate its *nPages=subChs*• *max_pageset.*

12) Calculate the total number of pages by

    total_page=(*max_pageset*-1)β + (the value of *promoted* of page-set *max_pageset*)+ (the value of *nPages* of page-set *max_pageset*)

13) Build the page-set schedule through the 2-dimensional array *ps_sch* with size $K\beta$ where *ps*[*i*][*j*], where $0 \leq i \leq K-1$ and $0 \leq j \leq \beta-1$, represents subchannel *j* of channel *i* or slot *j* of each block of channel *i* and holds the page-set number that is assigned to the subchannel. For each page-set *i* in *pagesetlist*, $1 \leq i \leq$ *max_pageset*, do the following:

    13a) For each subpage-set of page-set *i*, fill all the slots or subchannels belonging to this subpage-set in the corresponding channel with page-set *i.*

    13b) For each page-set *j* in *givelist* of page-set *i*, check all the *K* channels, subchannel by subchannel and backward from subchannel β -1, to find the

```
      last appropriate number of subchannels or slots belonging to this page-set $i$,
      and then reassign them to page-set $j$, which borrows the corresponding
      number of subchannels from page-set $i$
14) Build the page schedule according to the page-set schedule by mapping pages
    to each page-set in a strict sequential fashion.
    page_idx=0;
      Loop $i$ from 1 to max_pageset        //go through all the page-sets
          loop $j$ from 0 to $\beta$-1              //go through all the subchannels
              loop $k$ from 0 to $K$ - 1     //go through all the $K$ channels
                  if(ps_sch[$k$][$j$]==$i$)
                      for $m$=0 to $i$-1 //each subchannel broadcasts $i$
                                      //pages of page-set $i$
                      { page_idx++;
                        add page($k$, $j+m\bullet\beta$, $i\bullet\beta$, page_idx) to pagelist }
              Next $k$
          Next $j$
      Next $i$
15)   Output total_page and pagelist.
```

Figure 4.9 Algorithm for the HPB scheme

4.3 Simulation and Analysis of HPB

Figure 4.10 shows our simulation results for K=6 and the selected values of β between 100 and 10000. We can see that the normalized number of page-sets, i.e., (the total number of pages)/β, approaches the upper bound as β increases. The reason is that the proportion of promoted pages decreases and the residual bandwidth in Theorem 4.1, $K-\sum_{i=1}^{n}\frac{1}{i}$, is more likely to get some subchannels and contributes to the improvement of the result as β increases.

Table 4.1 shows the normalized number of page-sets, (the total number of pages)/β, achieved by HPB for the value of K from 3 to 7. The values of "GFDPB" and "RFS" are obtained by using the corresponding schedules of GFDPB (m=1) and RFS, respectively, as the initial version of rough block schedules. Since we need to add a header to each page during broadcasting, we can't let each page be too small;

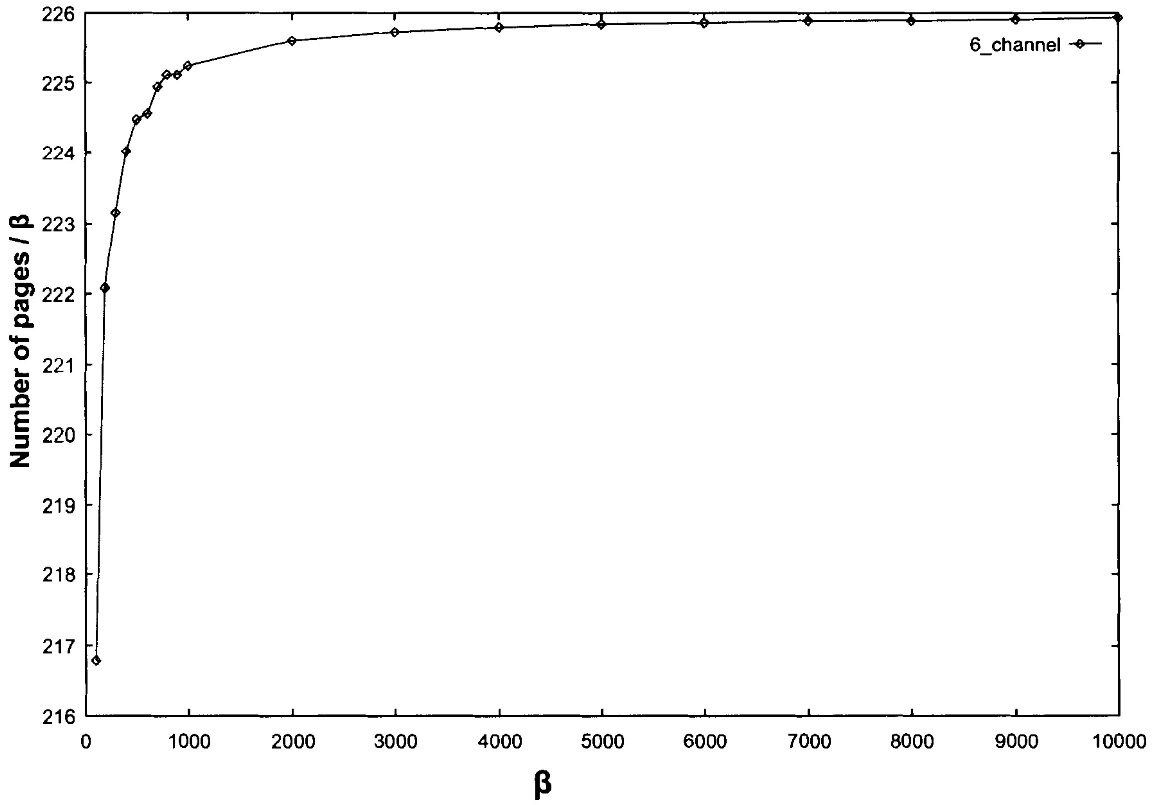


Figure 4.10 Normalized number of page-sets achieved by HPB

BW(*b*)	3	4	5	6	7
GFDPB	10.753	30.131	82.784	225.614	612.61
RFS	10.753	30.131	82.785	225.614	612.635
UB	10.781	30.155	82.828	226.009	615.215
β	60000	15000	5000	2000	1000

Table 4.1 Number of page-sets of HPB

otherwise, the header will consume too much bandwidth. To let each page remaining a reasonable record size, we can't let β grow too large. Here we limit the total number of pages to be no larger than 700,000. This means that each page can contain more than 7,200*5,000,000/(700,000*8)= 6,429 bytes of data given a two-hour movie and

5Megbits/second channel bandwidth. In the table, we choose the β for each case such that the number of pages packed is close to 700,000 pages. UB denotes the upper bound on the number of page-sets according to Theorem 4.1. We can see that the results of HPB based on the RFS schedule have little difference from those based on GFDPB schedules. All are very close to the upper bound if β is big.

BW(*b*)	3	4	5	6	7
HPB	334.790	119.478	43.486	15.956	5.876
Best Bound	400.00	128.571	46.753	17.061	6.315
RFS	400	144	49.315	17.910	6.372
Bar	400	144	49.315	17.734	6.417
GFDPB(*m*=1)	400	144	49.315	17.561	6.360
FD_LB	377.249	134.332	48.842	17.891	6.572
QHB(*m*=100)	360	124.137	43.902	16.071	5.892

Table 4.2 Average waiting time of HPB and other schemes

Table 4.2 compares the average waiting time, achieved by HPB and based on RFS schedules from Table 4.7, with other schemes for a two-hour movie. "Best Bound" shows the average waiting time achieved by Bar-Noy's non-perfect schedules (*K*=4, 5, 6) or from their modifications of the greedy algorithm using ad hoc method. Those are the shortest average waiting times of all schedules using segments of equal duration and channels of equal bandwidth. "FD_LB" shows the fixed-delay lower bound, the minimum waiting times that can be approached by schemes based on the fixed-delay policy such as FDPB, GFDPB, PHB and GEBB, according to Formula (2.5.2). "RFS" and "GFDPB" show the results of the RFS and GFDPB schemes, respectively. "Bar" shows the results of the multilevel splitting greedy algorithm. We can see none of the above three protocols or algorithms in the third group can guarantee an average waiting time better than the fixed-delay lower bound (FD_LB) in all cases, unless the number of channels is larger than 5 or 6. Moreover, their maximum waiting time is nearly 100% longer than FD_LB. Therefore, no currently known protocols or algorithms in the third group of broadcasting schemes can guarantee an average waiting time less than the fixed-delay lower bound, since the above three protocols or algorithms are the best in the third group in terms of the waiting time. We also show the results of QHB in Table 4.2, since QHB is the most efficient protocol in terms of clients' average waiting time in the other two groups of broadcasting protocols. We notice that the average waiting time of HPB is even shorter than that of QHB, although QHB can guarantee the lower average waiting time than the fixed-delay lower bound.

From the above simulation results, we can see HPB provides the lowest average waiting time of all currently known broadcasting protocols using the least number of channels given the server bandwidth, but it pays the price of using many more pages. For example, for K=6, GFDPB with m=1 partitions each video into 207 segments, whereas HPB needs 451,228 pages. Thus, the overhead cost of HPB is much more than that of other schemes.

We notice that the maximum waiting time of HPB is twice its average waiting time and much longer than those of FDPB or GFDPB. Since clients' requests come randomly, the fact that HPB's maximum waiting time is twice its average waiting time makes the treatment of clients highly variable and unfair.

4.4 Preloading Page-Set Broadcasting (PPSB)

To solve the problem of the maximum waiting time of HPB mentioned above, we propose a hybrid scheme, the PPSB scheme, which efficiently shortens the maximum waiting time of HPB by slightly compromising the average waiting time. In PPSB, clients need to wait half a block on average until the beginning of the next block, and then preload one time block before starting to play a video. Thus, in PPSB each page in the first page-set is broadcast once every two blocks. Its average waiting time is equal to 1.5*Dβ/total_page*. Its maximum waiting time is 2*Dβ/total_page*, and its minimum waiting time is *Dβ/total_page*. Thus, its maximum waiting time is 33.3% longer than its average waiting time.

For PPSB, we just need to make a slight change to the HPB algorithm in Figure 4.9. In step 1, the input file must be the schedule of GFDPB with *m*=2 and *pageset_num* must be "the last initial page-set number" instead of "the number of initial page-sets". In PPSB, we use the block period to indicate a page-set number, so the first page-set is page-set 2. Generally, the *i*-th page-set in PPSB is page-set *i*+1. Thus, in step 1 of Figure 4.9, a private member variable, *pageset,* of the class page-set is the block period of the corresponding page-set. A new theorem, Theorem 4.4, is required for PPSB based on modifying Theorem 4.1.

Theorem 4.4 Given *K* channels, let integer *n* satisfy

$$\frac{1}{2}+\frac{1}{3}+\bullet\bullet\bullet+\frac{1}{n+1}\leq K<\frac{1}{2}+\frac{1}{3}+\bullet\bullet\bullet+\frac{1}{n+2} \tag{4.4.1}$$

Then $n\beta+(K-\sum_{i=2}^{n+1}\frac{1}{i})(n+2)\beta$ is an upper bound on the number of pages of PPSB, and

$$n+(K-\sum_{i=2}^{n+1}\frac{1}{i})(n+2) \tag{4.4.2}$$

is an upper bound on the number of page-sets of PPSB. ■

Thus, given a video with a length of D seconds, $2D/(n+(K-\sum_{i=2}^{n+1}\frac{1}{i})(n+2))$ is a lower bound on the maximum waiting time and $1.5D/(n+(K-\sum_{i=2}^{n+1}\frac{1}{i})(n+2))$ is a lower bound on the average waiting time.

Formula (4.1.4) is changed to Formula (4.4.3).

$$\begin{aligned} &promoted(i) = \sum_{z=2}^{i-1}|PS(z)| - (i-2)\beta \qquad where\ i > 2 \\ &promoted(2) = 0 \end{aligned} \tag{4.4.3}$$

Formula (4.1.6.1) is changed to Formula (4.4.4.1).

$$|PS(i)| = (i-1)\beta - \sum_{z=2}^{i-1}|PS(z)| + promoted(i+1) \tag{4.4.4.1}$$

$$\geq (i-1)\beta - \sum_{z=2}^{i-1}|PS(z)| + lBound_\delta(i+1)$$

Accordingly, Formula (4.1.6) is changed to Formula (4.4.4).

$$|PS(i)| = i\left\lceil \frac{(i-1)\beta - \sum_{z=2}^{i-1}|PS(z)| + lBound_\delta(i+1)}{i} \right\rceil \text{ where } i > 2 \tag{4.4.4}$$

$$|PS(2)| = 2\left\lceil \frac{\beta + lBound_\delta(3)}{2} \right\rceil$$

Table 4.3 shows the average waiting time and the maximum waiting time achieved by PPSB and HPB. Their rough block schedules are all from GFDPB. The upper row of PPSB or HPB in the table shows the (average waiting time)/(maximum waiting time) in seconds for a two hour video, and the lower row shows the β value. Like in Table 4.1, we choose the β value so that the total number of pages of each schedule is less than and close to 700,000. We can see HPB's average waiting time is roughly 9.5% shorter than that of PPSB, but HPB's maximum waiting time is roughly 35% longer than that of PPSB. For K=5, by using PPSB instead of HPB, we increase the average waiting time by 4.57 seconds, but decrease the maximum waiting time by 22.89 seconds. We can say it is reasonable to sacrifice a small amount of average waiting time to win much more in maximum waiting time. By comparing Table 4.3 with Table 4.2, we can see PPSB still provides shorter average waiting time than the fixed-delay lower bound (FD_LB) and the shortest average waiting time of all the published broadcasting protocols in the third group. PPSB's maximum waiting time is not more than 32% longer than the fixed-delay lower bound (FD_LB), whereas HPB's maximum waiting time is nearly 90% longer than FD_LB. Although GFDPB (m=1), Bar and RFS may achieve their average waiting time shorter than the fixed-delay lower bound if the given number of channels is larger than 5 or 6, their corresponding maximum waiting time is nearly 100% longer than the fixed-delay lower bound.

BW(b)	3	4	5	6
PPSB	370.72/494.29	132.03/176.04	48.06/64.08	17.66/23.54
	23000	8000	3000	1000
HPB	334.79/669.58	119.48/238.96	43.49/86.97	15.95/31.91
	60000	15000	5000	3000

Table 4.3 Maximum and average waiting times of PPSB and HPB

We notice that QHB (see Table 4.2) achieves shorter average waiting times than PPSB. However, the superior performance of QHB in average waiting time comes at a price: achieving an average waiting time of 16.071 seconds for a two-hour video given a server bandwidth 6b requires dividing 224 fixed-sized segments into 2,519,678 smaller and smaller fragments and broadcasting them on 224 separate channels with decreasing bandwidth, which is rather impractical. Furthermore, the maximum waiting time of QHB is even longer than that of HPB.

	Non-perfect	HPB (β=3000)	PPSB (β=1000)	FDPB (m=1800)	GFDPB (m=1600)
Average	17.061	15.948	17.658	18.73	18.07
Max	34.123	31.897	23.544	18.73	18.07
Page# or Segment#	211	677189	611606	691787	637406

Table 4.4 Average and maximum waiting time for *K*=6

Table 4.4 compares the average waiting time achieved by PPSB for K=6 with other schemes or schedules for a two hour movie. Bar-Noy's non-perfect hand-tuned schedule is currently the best known schedule for average waiting time, but its maximum waiting time is twice its average waiting time. For K=6, compared with HPB, PPSB's maximum waiting time is 8.353 seconds shorter, but its average waiting time is 1.71 seconds longer. PPSB's minimum waiting time for K=6 is 11.772 seconds. Suppose that the actual waiting times of PPSB clients are distributed uniformly between the maximum waiting time and the minimum waiting time. Compared with the clients of GFDPB, the clients of PPSB enjoy shorter waiting times (up to (18.07-11.772) = 6.928 seconds shorter) with (18.07-11.772)/11.772=53.5% probability and wait longer (up to 5.474 seconds longer) with (23.544 -18.07)/11.771=46.5% probability. Waiting 5 seconds longer would be almost unnoticeable. Let us consider HPB for a comparison. In the worst case, clients of HPB need to wait 13.84 seconds longer than the clients of GFDPB; a period of 13.84 seconds would be noticeable. Thus, we can see that, while most of the time PPSB clients enjoy shorter waiting time than GFDPB clients, and even in the worst case, the maximum waiting time of PPSB would not be significant. Thus, at most times, the clients of PPSB would be happier than the clients of GFDPB.

In conclusion, PPSB is the only broadcasting protocol in the third group that guarantees its average waiting time less than the fixed-delay lower bound if β is big enough, while its maximum waiting time is only 1/3 longer than its average waiting time. From the probability point of view, PPSB provides a very desirable trade-off between average waiting time and maximum waiting time of all the known broadcasting protocols, given the server bandwidth.

Chapter Five
Conclusion and Future work

In this thesis, we first introduced different VOD protocols, and then focused on the broadcasting protocols, especially the third group of broadcasting protocols, which partition each video into a large number of small segments with equal size, and use time division multiplexing to multiplex the segments periodically into a small number of channels with bandwidth equal to the playback rate. We have proposed three new broadcasting protocols in this third group. GFDPB provides the lowest maximum waiting times of all protocols in the third group, and HPB provides the lowest average waiting times of all broadcasting protocols. PPSB provides the lowest average waiting times of all published protocols in the third group and provides a very desirable trade-off between average waiting time and maximum waiting time of all known protocols to date.

We presented two versions of GFDPB. GFDPB greatly improves the performance of the greedy algorithm by assigning the lowest priority to the split action that splits a leaf into a large number of child leaves with the same large window label. EGFDPB improves the performance of GFDPB when m is a prime number or has a big prime factor. We have shown that for $K>3$, the GFDPB result is almost the same as the EGFDPB result. We also presented a server multiplexing and client demultiplexing algorithm that can be applied to all the broadcasting protocols in the third group.

We have shown that no currently known broadcasting protocols in the third group can guarantee that their average waiting times are less than fixed-delay lower bound for any number of channels. By grouping pages into page-sets, and every β slots into a block, HPB can make the number of packed page-sets approach or even surpass the upper bound of RFS given the number of channels. However, HPB's maximum waiting time is still large compared with FDPB or GFDPB; therefore, we proposed the PPSB scheme to solve it.

We have shown that PPSB is the only broadcasting protocol that not only provides the average waiting time less than the fixed-delay lower bound but also guarantees its maximum waiting time is only 1/3 worse than its average waiting time. Other schemes that have a lower average waiting time than the fixed-delay lower bound, such as HPB and RFS, have their maximum waiting time 100% worse than their average waiting time. Therefore, PPSB provides much more fair and stable service for any client at any time than the above schemes.

Future research could be directed towards simplifying the algorithm of GFDPB, and making further improvements to GFDPB, or toward reducing the maximum waiting time of HPB. More work is needed to improve the performance of the protocols in this thesis when they are used to transmit variable bit-rate videos using a constant transmission rate, or when they are combined with multi-layer encoding videos and caching technique to provide multicast service through the internet.

Bibliography

[1] Y. H. Chang, D. Coggins, D. Pitt, and D. Skellem, "An open-system approach to video on demand, " *IEEE commun. Mag.*, vol. 32, pp.68-80, May 1994.

[2] D. Deloddere. W. Verbiest, and H. Verhille, "Interactive video on demand," *IEEE commun. Mag.*, vol.32, pp.82-88, May 1994.

[3] T. D. C. Little and D. Venkatesh, "Prospects for interactive video-on-demand," *IEEE Multimedia*, vol. 1, pp. 14-24, Mar. 1994.

[4] L. Gao and D. Towsley, "Efficient schemes for broadcasting popular videos," in *Proc. Int. Workshop Network and Operating Syst. Support for Digital Audio and Video*, pp.317-329, Aug. 1998.

[5] R. Rejaie, M. Handley, H. Yu, and D. Estrin, "Proxy caching mechanism for multimedia playback streams in the internet," In *Proc. 4th Int. Web caching Workshop*, 1999.

[6] E. Bommaiah, K. Guo, M. Hofmann, and S. Paul, "Design and implementation of a caching system for streaming media over the Internet," *IEEE Real-Time Technology and Application Symposium*, 2000.

[7] Dong-Hoon Nam and Seung-Kyu Park, "Adaptive multimedia stream service with intelligent proxy," Int. *Conf. on Information Networking*, 2001.

[8] Chen-Lung Chan, Te-Chou Su, Shih-Yu Huang, and Jia-Shung Wang, "Cooperative proxy scheme for large-scale VoD systems," *Parallel and Distributed Systems*, 2002. *Proc. Ninth International Conference*, pp. 404 – 409, Dec. 2002.

[9] http://www.ncube.com/pressroom/downloads/npvr-white-paper.pdf.

[10] A. Dan D. Sitaram, and P. Shahabuddin, "Scheduling policies for an on-demand video server with batching," *Proc. ACM Multimedia*, San Franciso, CA, pp. 15-23, Oct. 1994.

[11] A. Dan, D. Sutaram, and P. Shahabuddin, "Dynamic batching policies for an on-demand video server," *Multimedia Systems*, 4(3):112-121, June 1996.

[12] HP website download file: http://h71028.www7.hp.com/enterprise /downloads /VOD_datasheet_6-05-03_v2.pdf.

[13] Kasenna, Inc. Website: http://www.kasenna.com/newkasenna/solutions/ Kasenna.MediaBaseXMPv70.Datasheet.08.15.03c.pdf.

[14] David P. Anderson, "Metascheduling for continuous media", *ACM Transactions on Computer Systems*, Vol. 11, No. 3, pp.226-252, Aug.1993.

[15] A. Dan, D. Sitaram, and P. Shahabuddin, "Scheduling policies for an on-demand video server with batching," in *Proc. ACM Multimedia*, Florence, Italy, pp. 15-23, 1994.

[16] A. Dan, D. Sitaram, and P. Shahabuddin, "Dynamic batching policies for an on-demand video server", *Multimedia Systems*, 4(3): June 1996.

[17] C. C. Aggarwal, J. L. Wolf and P. S. Yu, "On optimal batching policies for video-on-demand storage servers," in *Proc. IEEE int. Conf. Multimedia Computing and Systems*, June 1996, pp. 253-258.

[18] E. L. Abram-Profeta and K. G. Shin, "Scheduling video programs in near video-on-demand systems", *In Proc. ACM Multimedia'97*, Nov. 1997.

[19] K. C. Almeroth and M. Ammar, "The interactive multimedia jukebox (IMJ): A new paradigm for the on-demand delivery of audio/video", *In Proc. 7th WWW conference*, Brisbane, Australia, April 1998.

[20] S. H. G. Chan, F. Tobagi, and T. M. Ko, "Providing on-demand video services using request batching", in *IEEE Int. Conf. Communication*, vol.3, pp.1716-1722, 1998.

[21] W.K.S. Tang, E.W.M. Wong, S. Chan, and K.-T. Ko, "Optimal video placement scheme for batching VOD services", *Broadcasting, IEEE transactions on*, vol. 50, issue 1, pp. 16-25, Mar. 2004.

[22] L. Gao and D. Towslay, "Supplying instantaneous video-on-demand services using controlled multicast," *IEEE Multimedia*, pp.117-121, June 1999.

[23] K. A. Hua, Y. Cai, and S. Sheu, "Patching: a multicast technique for true video-on-demand services", ACM Multimedia, pp.191-200, Sept. 1998.

[24] S. Sen, L. Gao, J. Rexford, and D. Towsley, "Optimal patching schemes for efficient multimedia stream", *In Proc. of IEEE NOSSDAV*, NJ, USA, June 1999.

[25] Y. Cai, K. A. Hua, and K. Vu, "Optimizing patching performance", *In Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, CA, pp. 204-215, Jan. 1999.

[26] D. Eager, M. Vernon, and J. Zahorjan, "Optimal and efficient merging schedules for video-on-demand servers," in *Proc. 7th ACM Int. Multimedia Conf.* (ACM MULTIMEDIA'99), 1999.

[27] D. L. Eager, M. K. Vernon, and J. Zahorjan, "Minimizing bandwidth requirements for on-demand data delivery," *IEEE Trans. Knowledge and Data Engineering*, Sept./Oct. 2001.

[28] D. Guan and S. Yu, "A two-level patching scheme for video-on-demand delivery", *IEEE Transactions on Broadcasting*, vol. 50, No. 1, Mar. 2004.

[29] E. G. Coffman Jr., P. Jelenkovic' and P. Mom∞ilovic', "The Dyadic stream merging algorithm", *In J. Algorithms*, 43(2002), 120-137.

[30] A. Bar-Noy, J. Goshi, and R. E. Ladner, "Off-line and on-line guaranteed start-up delay for media-on-demand with stream merging", in *Proc. 15th annual ACM symposium on Parallel algorithm and architectures*, San Diego, California, USA, session: Networks II, pp.164-173, 2003.

[31] A. Bar-Noy and R. E. Ladner, "Competitive on-line stream merging algorithms for media-on-demand", In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* 364-373, Jan. 2001.

[32] K. S. Tang, K. T. Ko, S. Chan, and E.W. M. Wong, "Optimal file placement in VOD system using genetic algorithm," *IEEE Trans. Industrial Electronics,* vol. 48, no. 5, pp. 891–897, Oct. 2001.

[33] W. K. S. Tang, E. W. M. Wong, S. Chan, and K. –T. Ko, "Optimal video placement scheme for batching VOD service", *IEEE Transaction on Broadcasting*, vol. 50, issue 1, Mar. 2004, pp.16-25.

[34] D. L. Eager and M. K. Vemon, "Dynamic skyscraper broadcasts for video-on-demand," in *Proc. 4th Int. Workshop on Multimedia Information systems (MIS'98)*, Istanbul, Turkey, pp. 18–32, Sept. 1998.

[35] D. L. Eager, M. C. Ferris, and M. K. Vernon, "Optimized regional caching for on demand data delivery," in *Proc. IS&T/SPIE Conf. Multimedia Computing and Networking 1999 (MMCN'99)*, San Jose, CA, pp. 301–316, Jan. 1999.

[36] S. W. Carter and D. D. E. Long, "Improving videoon-demand server efficiency through stream tapping", *Proceedings of the 6th International Conference on Computer Communications and Networks*, pp. 200-207, Sep. 1997.

[37] S. W. Carter and D. D. E. Long, "Improving bandwidth efficiency on video-on-demand servers", *Computer Networks and ISDN Systems*, 30(1–2):99–111.

[38] C. C. Aggarwal, J. L. Wolf, and P. S. Yu, "On optimal piggyback merging policies for video-on-demand systems," in *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, pp. 200–209, 1996.

[39] L. Golubchik, J. C. S. Lui, and R. R. Muntz, "Reducing I/O demand in ideo-on-demand storage servers," in *Proc. ACM SIGMETRICS Conf. measurement and Modeling of Computer Systems (SIGMETRICS'95)*, pp. 25–36, 1995.

[40] L. Golubchik, J. C. S. Lui, and R. R. Muntz, "Adaptive piggybacking: A novel technique for data sharing in video-on-demand storage servers," ACM Multimedia Systems Journal, vol. 4, no. 3, pp. 140–155, 1996.

[41] S. W. Lau, J. C. S. Lui, and L. Golubchik, "Merging video streams in a multimedia storage server: Complexity and heuristics," *ACM Multimedia* Systems Journal, vol. 6, no. 1, pp. 29–42, 1998.

[42] T. Chiueh and C. Lu, "A periodic broadcasting approach to video-on-demand service", *Int. Soc. Optical Eng.,* vol. 2615, pp.162-169, Oct. 1995.

[43] S. Viswanathan and T. Imielinski, "Pyramid broadcasting for video-on-demand service," in *Proc. SPIE Conf. Multimedia Computing and* Networking (MMCN'95), pp. 66–77, 1995.

[44] S. Viswanathan and T. Imielinski, "Metropolitan area video-on-demand service using pyramid broadcasting," *ACM Multimedia Systems Joumal*, vol. 4, no. 3, pp. 197–208, 1996.

[45] C. C. Aggarwal, J. L. Wolf, and P. S. Yu, "A permutation-based pyramid broadcasting scheme for video-on-demand systems", in *Proc. IEEE Int. Conf. Multimedia Computing and Systems*, pp. 118-126, June 1996.

[46] K. A. Hua and S. Sheu, "Skyscraper Broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems", in *Proc. Of the ACM SIGCOMM'97 Conference*, Connes, France, pp.89-100, Sept. 1997.

[47] L. Juhn and L.Tseng, "Fast data broadcasting and receiving scheme for popular video service", in *IEEE Transactions on Broadcasting*, 44(1):100-105, Mar. 1998.

[48] A. Hu, I. Nikolaidis, and P. van Beek, "On the design of efficient video-on-demand broadcast schedules", in *Proc. 7th Int'l Symp. On Modeling Analysis and Simulation of Computer and Telecommunication Systems* (MASCOTS'99), pp.262-269.

[49] L. Juhn and L. Tseng, "Harmonic broadcasting for video-on-demand service", *IEEE Trans. On Broadcasting*, 43:268-271, Sep. 1997.

[50] P. Mundur, R. Simon, and A. K. Sood, "End-to-end analysis of distributed video-on-demand systems", in *IEEE Transactions on Multimedia,* vol. 6, No.1, Feb. 2004.

[51] A. Hu, "Video-on-demand broadcasting protocols: a comprehensive study", in *Proc. IEEE INFOCOM 2001, 20th Annual Joint Conf. of IEEE Computer and Communications Societies*, vol. 1, 22-26, pp. 508-517, April 2001.

[52] Z. Y. Yang, L. S. Juhn, and L.M. Tseng, "On optimal broadcasting scheme for popular video service", *IEEE Trans. Broadcast*, vol. 45, pp. 318-322, Sept. 1999.

[53] J. F. Paris, S. W. carter, and D. D. E. Long, "Efficient broadcasting protocols for video on demand", in *6th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'98)*, pp. 127-132, July 1998.

[54] J. F. Paris, S. W. carter, and D. D. E. Long, "A low bandwidth broadcasting protocol for video on demand", in *Proc. 7th ICCCN Conf.*, pp. 690-697, Oct. 1998.

[55] J. F. Paris, S. W. carter, and D. D. E. Long, "A hybrid broadcasting protocol for video on demand", in *Proc. 1999 Multimedia Computing and Networking Conference*, San Jose, CA, pp.317-326, Jan. 1999.

[56] J. F. Paris, "A simple low-bandwidth broadcasting protocol for video on demand", in *Proc. 87th Int. ICCCN Conf.*, pp.690-697, Oct. 1999.

[57] J.-F. Pâris, "A Fixed-Delay Broadcasting Protocol for Video-On-Demand", in *Proc. 10th International Conference on Computer Communications and Networks, 2001*, pp. 418-423, 2001.

[58] A. Bar-Noy and R. E. Ladner, "Windows scheduling problems for broadcast systems", in *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* 433-442, Jan. 2002.

[59] Y. C. Tseng, M. H. Yang, and C. H. Chang, "A recursive frequency-splitting scheme for broadcasting hot videos in VOD service", *IEEE Transactions on Communication,* vol. 50, issue 8, pp.1348-1355, Aug. 2002.

[60] A. Bar-Noy, R. E. Ladner, and T. Tamir, "Scheduling technique for media-on-demand", in *Proc. 14th annual ACM-SIAM symposium on Discrete algorithms*, Baltmore, Maryland, session 11C, pp. 791-880.

[61] E.M. Yan and T. Kameda, "An efficient VOD broadcasting scheme with user bandwidth limit", in *Proc. SPIE/ACM Conf. on Multimedia Computing and Networking*, vol. 5019, Santa Clara, CA, pp.200-208, Jan. 2003.

[62] O. Bagouet, K. A. Hua, and D. Oger, "A Periodic Broadcast Protocol for Heterogeneous Receivers," in *Proc. of SPIE Conference on Multimedia Computing and Networking (MMCN 2003)*, pp. 220-231, January 29-31, 2003.

[63] J. Y. B. Lee, "On a unified architecture for video-on-demand services," *IEEE Trans. Multimedia*, vol. 4, no. 1, pp. 38–47, Mar. 2002.

[64] W. F. Poon, K. T. Lo, and J. Feng, "A hybrid delivery strategy for a video-on-demand system with customer reneging behavior," *IEEE Trans. Broadcast.*, vol. 48, no. 2, pp. 140–150, June 2002.

[65] T. Kameda, and Y. Sun, "Optimal truncated-Harmonic windows scheduling for broadcast systems", in *Proc. ACM SAC04,* Nicosia, Cyprus, March 2004.

[66] A. Bar-Noy, A. Nisgav and B. Patt-Shamir, "Nearly optimal perfectly-periodic schedules", In *Proc. 20th annual ACM symposium on Principles of distributed computing,* Newport, Rhode Island, United States, pp. 107-116, Aug. 2001.

[67] T. Kameda and Y. Sun, "Group windows scheduling through windows scheduling", submitted to *INFOCOM* 2005.