Multimedia Systems (1998) 6:29–42

Multimedia Systems

Merging video streams in a multimedia storage server: complexity and heuristics

Siu-Wah Lau[1], **John C.S. Lui**[2], **Leana Golubchik**[3,⋆]

[1] Department of Computer Science and Engineering, The Chinese University of Hong Kong; e-mail: swlau@cs.cuhk.edu.hk
[2] Department of Computer Science and Engineering, The Chinese University of Hong Kong; e-mail: cslui@cs.cuhk.edu.hk
[3] Department of Computer Science, Columbia University, New York, NY 10027, USA; e-mail: leana@cs.columbia.edu

Abstract. Due to recent advances in network, storage and data compression technologies, video-on-demand (VOD) service has become economically feasible. It is a challenging task to design a video storage server that can efficiently service a large number of concurrent requests on demand. One approach to accomplishing this task is to reduce the I/O demand to the VOD server through data- and resource-sharing techniques. One form of data sharing is the **stream-merging approach** proposed in [5]. In this paper, we formalize a static version of the stream-merging problem, derive an upper bound on the I/O demand of static stream merging, and propose efficient heuristic algorithms for both static and dynamic versions of the stream-merging problem.

Key words: Multimedia storage systems – Video-on-demand – Stream merging – I/O stream sharing – Complexity

1 Introduction

Recent advances in networking technologies and vast improvements of storage systems have made it feasible to provide multimedia on-demand services, such as news distribution, advertisement, library information systems, and movies-on-demand. Consequently, the area of multimedia storage systems has received a great deal of attention in the past few years.

Early research works [2,10,12] concentrated on the study of multimedia storage systems which support the retrieval of multimedia objects at *peak* display bandwidths (bits/s); for example, assuming that the display bandwidth of the object is fixed at $B_{display}$ Mbps throughout the duration of the display, the storage server retrieves that object using $B_{display}$ Mbps of disk I/O bandwidth. This approach may be applicable for video objects whose average display bandwidth requirements are close to their peak display bandwidth requirements. (The variance of a video object's bandwidth requirement is a function of the compression technique used and the actual content of that object.) Recent works consider video-on-demand (VOD) architectures which can support variable display bandwidths. For example in [7–9], the authors propose novel techniques for supporting variable display bandwidth for a disk-based storage architecture and for a hierarchical storage architecture.

Part of the challenge of designing efficient VOD servers is due to the large storage and bandwidth requirements of video objects. For example, a 120-min MPEG-I video requires 1.5 Mbits/s of display bandwidth and 1.3 GB of storage [6]. However, the quality of MPEG-I video is at best VHS quality and is certainly lower quality than broadcast television. Higher quality compressed video, such as MPEG-II or compressed HTDV video [1], requires display bandwidths of 4 to 20 Mbits/s. The storage requirements of video objects usually precludes them from being stored in main memory. Video objects have to be stored on magnetic disks or tertiary storage devices such as robotic tape libraries. A video object is displayed by scheduling an I/O stream where the data is read from an appropriate storage device or a set of storage devices and delivered to a display unit.

There are many approaches to improving the efficiency of a VOD storage system; these include improvements in data layout techniques, disk-scheduling algorithms, etc. In this paper we concentrate on the the delivery of *popular* objects in a VOD system. That is, we expect there to be a skew in the distribution of access frequencies of the video objects. Moreover, we expect that a small subset of objects would be accessed very frequently, and the rest of the objects would be accessed infrequently; such an access pattern would, for instance, be accurate for a movie server where a small subset of popular movies (perhaps for that week) is accessed simultaneously by many users. In such a system, it is very likely that the I/O bandwidth is the critical resource which contributes to a large fraction of the system response time. We define system response time as the time between a request's arrival to the storage server and the time when we can start streaming the bits of the requested video from the disk storage system to the network. Although disk storage costs are decreasing rapidly, bandwidth costs are not decreasing nearly as quickly; this is partly due to the fact that most storage improvements are due to increases in the

⋆ *Current address:* Department of Computer Science at University of Maryland, College Park, MD 20742, USA; e-mail: leana@cs.umd.edu
Correspondence to: L. Golubchik

number of tracks per inch, rather than the number of bits per track (and, the number of bits per track plus the revolutions per minute determine the bandwidth capability of a disk). One way to improve the system response time for delivery of popular objects is to carefully manage the I/O bandwidth of the VOD server and "share" it among requests for the same object. The following approaches to reducing the aggregate I/O bandwidth requirements of popular objects have been proposed in the past:

- *Batching*: the storage server polls the request queue periodically and serves requests, for the same object, that have accumulated in the queue, using a single I/O stream, e.g., [3].
- *Buffering*: two or more successive requests for the same object can be served by temporarily holding the data, retrieved by a single I/O stream, in the main memory buffers, where the first request is serviced using the I/O stream and successive requests are serviced from the main memory buffers, e.g., [4, 11].
- *Adaptive piggybacking or stream merging*: the display rates of requests in progress (for the same object) are dynamically adjusted until their corresponding I/O streams can be "merged" into one [5].

Batching, buffering, and stream merging can serve a group of requests, for the same object, using a single I/O stream. Stream merging differs from batching in that it groups requests dynamically, while displays are in progress, so that no additional latency is experienced by the user. On the other hand, batching requires that the displays of requests of the same group start at the same time and hence contributes to additional delays in the system. Note that, the reduction in I/O bandwidth demand due to stream merging is not quite as high as in the case of batching, since it takes some time to merge the streams (and no I/O bandwidth savings are accomplished during that time). Thus, the trade-off between the two approaches is in balancing the latency for starting service of a request and the amount of I/O bandwidth saved. It is important to point out that all three approaches can be combined, as mentioned in [5].

In this paper, we concentrate on the stream-merging approach because of its effectiveness as reported in [5]. First, we briefly elaborate on the motivation and feasibility of this approach; a more detailed and more formal description of stream merging is given in Sect. 2. The stream-merging approach is motivated by the fact that it is possible to time compress or time expand a video object by a small percentage (e.g., 5%) without it being perceptible by the user, i.e., it is possible to alter the duration of an object's display without affecting its (perceptible) quality. Similarly, the duration of an audio object can be altered, for instance, using techniques such as audio pitch correction. Ample evidence exists to support the above-stated claims. In [5], the authors give a detailed explanation of the feasibility of such video and audio alteration techniques. In the interests of brevity, we only give a brief example here and refer the reader to [5] for a more detailed explanation. Consider, for instance, airing of movies on television. It is common practice in the television industry to time compress a movie for the purpose of increasing the number of commercial advertisements shown. For instance, when the movie "Amadeus" was shown on television, its duration was altered by 3%.

There are two approaches to actually constructing the altered stream of frames to be transmitted to a display station. They are as follows:

- *Online approach*: the altered version of the object can be created online or on the fly. An I/O stream retrieves the original object, which is then time expanded or compressed by the server[1]. The "derived" object is transmitted to the display unit. In this case, the I/O bandwidth required varies with the display rate used. There are two possible disadvantages of the online alteration: (1) the data layout on disks is often tuned to one delivery bandwidth, and having to support multiple bandwidths can complicate scheduling and/or require additional buffer storage, and (2) specialized hardware may be required to be able to produce the altered version in real time.
- *Offline approach*: the altered version of the video can be created offline and stored on the disk in addition to the original version. An I/O stream retrieves an appropriate version of the object (be it original, time expanded, or time compressed), which is then directly transmitted to the display unit. An obvious disadvantage of this approach is the additional disk storage required.

Whether we use the online or offline approach, the stream-merging technique requires the capability of transmitting a video object at several different display rates. There are multiple ways of attacking this problem, for instance, by using techniques similar to the ones proposed in [7–9]. Note that the details of supporting multiple display rates depend on the particular VOD server architecture used. We do not consider a specific architecture here since we are interested in developing a general technique for reducing I/O bandwidth demand using stream merging. Therefore, in this paper, we do not address the details of supporting multiple display rates.

In [5], the authors proposed several heuristic merging policies and analyzed the subsequent performance improvements using analytical models. However, many questions about the stream-merging approach remain unanswered, for example, determining an optimal stream-merging policy, as well as the maximum (or minimum) achievable performance improvements of the approach. We intend to address some of these questions in this paper. The contributions of this work are as follows: (1) we formalize a version of the stream-merging problem and derive an upper bound on its I/O demand, and (2) we propose two novel stream-merging algorithms, which result in significant I/O demand reductions.

The organization of the paper is as follows. In Sect. 2, we formalize the *static* version of the stream-merging problem and describe the *dynamic* version of the stream-merging problem. In Sect. 3, we propose a heuristic algorithm for solving the static stream-merging problem and derive the properties of optimal solutions of the static stream-merging problem as well as the maximum I/O demand for merging n streams. We propose a novel dynamic stream-merging algorithm, Equal-Split, in Sect. 4. Section 5 presents per-

[1] We give a more precise definition of "time expansion" and "time compression" of video objects in Sect. 2.

formance analysis of algorithm Equal-Split. In Sect. 6 we compare performance of several dynamic stream-merging algorithms. Our conclusions are given in Sect. 7.

2 Problem definition

The stream-merging approach [5] initiates an I/O stream (or simply stream) for each request. Then, the display rates of the streams, corresponding to requests for the same object, are adjusted until the streams output the same data at the same time. At this point the I/O streams are merged into a single stream and the corresponding requests share this single I/O stream.

We assume that the storage server transmits frames to the display units at a constant frame rate, e.g., the NTSC standard requires that the display units display at 30 frames per second (fps). The stream-merging approach is viable if the storage server can *time compress* or *time expand* some sequence of original object frames. For example, we can time expand a sequence of original object frames by adding one additional frame to every 19 original object frames. Then, a display unit displays $30 \times \frac{19}{20} = 28.5$ original object frames per second. Similarly, we can time compress a sequence of original object frames by removing frames. A sequence of original object frames is time compressed if the display time of this sequence is somehow shortened as compared to its normal display time. Similarly, a sequence of original object frames is time expanded if the display time of the sequence is longer than its normal display time. More formally, time expansion and time compression can be defined as follows. Let $f_1, \ldots, f_k$ be a sequence of original object frames. Let $f'_1, \ldots, f'_m$ be a sequence of frames which are derived from $f_1, \ldots, f_k$ and are fed to the display unit. The sequence of original frames is time expanded if $m > k$; it is time compressed if $m < k$. (The two possible approaches to producing time-expanded or time-compressed versions of an object are discussed in Sect. 1.)

We define the display rate alteration ratio of an I/O stream as follows. If the I/O stream is a sequence of consecutive original object frames, then the display rate alteration ratio of the I/O stream is equal to 1. If the I/O stream is a stream of frames derived from original consecutive object frames, then the display rate alteration ratio of the I/O stream is equal to the number of consecutive original object frames required to derive one frame of the I/O stream. For example, if the frames of an I/O stream are derived by removing 1 out of every 21 original consecutive object frames, then the display rate alteration ratio of that I/O stream is equal to $\frac{21}{20} = 1.05$.

The display rate of an I/O stream is defined as the number of frames output per second $\times$ the display rate alteration ratio of the I/O stream. Therefore, the display rate of an I/O stream is a measure of how fast the I/O stream gets through the content of the "original" video. The *effective display rate* of a display unit is defined as the display rate of the I/O stream being transmitted to the display unit. Note that the frames of an I/O stream may be time expanded or compressed before transmission to the display unit, if the online approach to display rate alteration is used.

Let S_n be the normal display rate in frames per second (fps). Let Δ_+ and Δ_- be the maximum fraction of the normal display rate by which a stream can be sped up or slowed down, respectively, i.e., a stream is constrained to output at a display rate between $(1 - \Delta_-)S_n$ and $(1 + \Delta_+)S_n$ frames per second (fps). The display rate of a request is defined as the display rate of a stream. For convenience of further discussion, we introduce the following definitions.

Definition 1. *The playback point of a stream s at time t, $p_s(t)$, is the current position (in seconds) in the object's display of stream s at time t.*

Definition 2. *Two streams are said to be "synchronized", if playback points of the streams are the same.*

Note that two streams can be merged into a single stream when they are synchronized (at this point, system resources can be saved). Our goal is to design a general algorithm for synchronizing streams in an optimal way for VOD systems[2]. The amount of system resources required by each stream is a function of the system architecture, i.e., communication protocols used, storage and retrieval methods used, etc. Since we are concerned with the delivery of popular objects, it is reasonable to assume that I/O demand is the critical resource. Note that, the stream-merging approach should be applicable to reducing demand on other system resources, e.g., the communication network bandwidth. However, the specific tradeoffs associated with applying this approach to another resource may differ from those we consider here, in the context of I/O bandwidth demand.

In this paper, we consider two possible versions of the stream-merging problem, namely: (1) the *static* version and (2) the *dynamic* version. In both cases, our goal is to merge streams corresponding to requests for the same object, in order to reduce the aggregate I/O bandwidth demand on the system. In the static case, we consider a single group of streams such that the membership of the group is fixed throughout the stream-merging process, i.e., no new streams can be initiated (which would correspond to an arrival of a new request) and no stream can be terminated (which would correspond to an end of an object's display). Thus, in the static merging problem, synchronization decisions can be made under a "complete information" assumption. The dynamic case differs from the static case in that new streams can be initiated and existing streams can be terminated. As a result, synchronization decisions have to be made "on the fly", without having full information about future arrivals. More formally, the static and dynamic merging problems can be defined as follows:

1. *Static stream-merging problem*: Given a set of streams of a video object, which has an infinite display time, and the corresponding playback points of these streams, find an optimum way to merge them into a single stream, where the objective is to minimize the total I/O cost (in bits) incurred during the synchronization process[3].

[2] For the remainder of the paper, we use the terms "synchronizing" and "merging" interchangeably.

[3] The assumption of infinite display time of a video object is used to prevent any stream termination due to reaching the end of an object's display. In later sections, we will show that one way to solve the dynamic

2. *Dynamic stream-merging problem*: The problem of minimizing the I/O cost (in bits) of retrieving data through merging of streams corresponding to requests for the same object in a VOD system in which
   – a new stream of a video object can be initiated due to a request arrival,
   – a stream can terminate due to reaching the end of an object's display, and
   – the request arrival process is stochastic.

In the remainder of this section and in Sect. 3, we concentrate on the static stream-merging problem. Before proceeding to characterize the stream-merging problem further and describing our algorithms, we make the following observation about the display rate adjustment decisions. The sooner merging (in an object's display) occurs, the more I/O bandwidth can be conserved and used by the storage system to serve other requests. Hence, we limit our algorithms to consider the slowest display rate, $S_{min} = (1 - \Delta_-)S_n$, the normal display rate, S_n, and the fastest display rate, $S_{max} = (1 + \Delta_+)S_n$; the corresponding I/O bandwidths are C_{min}, C_n, and C_{max}, respectively. The relative values of C_{min}, C_n, and C_{max} depend on the display rate alteration technique used. We restrict the rest of the discussion in this paper to the class of display rate alteration approaches where $\forall S' \in [S_{min}, S_{max}]$, $C_{min} \leq$ the I/O bandwidth corresponding to display rate $S' \leq C_{max}$.

Suppose we would like to synchronize n streams corresponding to requests for the same video object. Let $\{s_1, \ldots, s_n\}$ be this set of streams. Without loss of generality, we assume that (1) $\forall i, j$ if $i < j$, then $p_{s_i}(0) > p_{s_j}(0)$ (i.e., s_1 is the leading stream and s_n is the trailing stream), (2) the synchronization of streams $s_1, \ldots, s_n$ begins at time $t = 0$, and (3) if two streams s_i and s_j are merged and $i < j$, the I/O resources used by s_j are released and the requests being served by s_j are served by s_i.

For the purpose of solving the static stream synchronization problem, we are only interested in the differences between the playback points of the streams, where the objective is to reduce the differences between the playback points of all streams to zero, i.e., to come to a point where each stream outputs the same video data at the same time. In the following definition, we introduce the concept of a relative playback point as a measure of the relative position of a stream.

Definition 3. *The relative playback point of a stream s_i at time t, $r_{s_i}(t)$, is $p_{s_i}(t) - t - p_{s_n}(0)$.*

Note that the relative playback point of stream s_i changes when s_i is sped up or slowed down. For example, if s_i is sped up by 5% of the normal display rate, the relative playback point of s_i increases by 0.05 seconds per second.

We can represent the process of synchronization of streams by a synchronization tree, which can be defined as follows.

stream-merging problem is by: (1) partitioning the streams of the same video object into disjoint sets such that all the streams in each set can be merged into a single stream without any stream terminations and (2) applying a static stream-merging algorithm to merge the streams in each set.

Definition 4. *A synchronization tree T is a tree in which the root node represents the final stream resulting from synchronization of n (original) streams, the leaf nodes represent the n (original) streams, $s_1, \ldots, s_n$, and each internal node represents a stream derived from the synchronization of its child nodes (streams).*

Let $C_s(t)$ be the I/O bandwidth demand (in bits/s) of stream s at time t. If stream s has been discarded at time t due to merging with another stream, then $C_s(t') = 0\ \forall t' > t$. Then, we can define the cost of a synchronization tree as follows.

Definition 5. *The cost of a synchronization tree T, corresponding to a set of n (original) streams, $\{s_1, \ldots, s_n\}$, is defined as*

$$cost(T) = \sum_{i=1}^{n} \int_0^P C_{s_i}(t)\, dt\ ,$$

where P is the time required for the synchronization.

As will become more apparent in Sect. 3, our goal of solving the static stream-merging problem will translate into the goal of constructing an optimum synchronization tree.

Definition 6. *A synchronization tree T is said to be optimal if $cost(T') \geq cost(T)\ \forall$ synchronization trees T' with leaves $s_1, \ldots, s_n$.*

3 Complexity of static stream merging

In this section, we characterize the properties of an optimal synchronization tree as well as propose an efficient algorithm for solving the static stream-merging problem, based on the idea of constructing a synchronization tree. We restrict the class of algorithms considered in this section, to the algorithms with the following property: once a decision to merge two streams has been made, their display rates remain unchanged until the completion of the merging process.

In an optimal synchronization tree, the leading stream s_1 is always slowed down by a fraction of Δ_- and the trailing stream s_n is always sped up by a fraction of Δ_+. The synchronization process completes when the trailing stream and the leading stream are synchronized. Hence, we have the following lemma.

Lemma 1. *The time, P, to synchronize the set of streams $\{s_1, \ldots, s_n\}$ is $\frac{p_{s_1}(0) - p_{s_n}(0)}{\Delta_+ + \Delta_-}$.*

Proof. At any time $t \in [0, P]$, the trailing stream s_n has a display rate of $(1 + \Delta_+)\times$ the normal display rate, and the leading stream s_1 has a display rate of $(1 - \Delta_-)\times$ the normal display rate. Hence, the trailing stream and the leading stream move towards each other at a rate of $\Delta_+ + \Delta_-$ seconds per each second of display. Therefore, it takes $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{\Delta_+ + \Delta_-}$ seconds to synchronize all the streams.

Lemma 2. *An optimal synchronization tree is a binary tree.*

Proof. From Definition 4, each internal node, including the root of the tree, has at least two children. Suppose there exists an optimal cost tree T which contains a node with three children s_a, s_b, and s_c. Without loss of generality, we assume that $r_{s_a}(t) < r_{s_b}(t) < r_{s_c}(t)$. This situation is illustrated in Fig. 1a. Let the display rates of s_a, s_b, and s_c be S_a, S_b, and S_c, respectively. Note that $S_a > S_b > S_c \geq S_{min}$, because the three streams are merged at the same time. Let the I/O bandwidths of s_a, s_b, and s_c be C_a, C_b, and C_c, respectively. Let P_1 be the time of synchronization of streams in T_1. The cost of subtree T_1, covering s_a, s_b, and s_c in Fig. 1a, is

$$P_1(C_a + C_c) + P_1 C_b \ ,$$

where $C_{min} \leq C_b \leq C_{max}$. The first term indicates the I/O cost for merging s_a and s_c, while the second term indicates the I/O cost for merging s_b.

We can now show that a subtree T_2 with a cost lower than that of T_1 can be constructed to cover s_a, s_b, and s_c by synchronizing streams s_a and s_b first [4], followed by synchronization of streams s_a and s_c. In this case, the display rate of s_b is S_{min}. This situation is illustrated in Fig. 1b. Let P_2 be the synchronization time of s_a and s_b. Since s_a and s_b are merged first, $P_2 < P_1$. The cost of subtree T_2, covering s_a, s_b, and s_c in Fig. 1b, is

$$P_1(C_a + C_c) + P_2 C_{min} \ .$$

The first term represents the I/O cost for merging s_a and s_c, while the second term indicates the I/O cost for merging s_b. Since $C_{min} \leq C_b$ and $P_2 < P_1$, the cost of subtree T_2 is less than that of subtree T_1. Therefore, the synchronization tree in Fig. 1b has a lower cost than the synchronization tree in Fig. 1a. Hence, an optimal tree cannot have a node with three children. Similarly, we can show that an optimal tree cannot have a node with four or more children by merging streams in a pair-wise manner. The result follows.

The cost of stream synchronization depends on the order in which the streams are synchronized. For example, suppose $\Delta_+ = \Delta_- = 0.05$, $C_n = C_{min} = C_{max} = 1.5$ Mbits/s, and there are four streams s_1, s_2, s_3, and s_4, where $p_{s_1}(0) = 15$ s, $p_{s_2}(0) = 9.5$ s, $p_{s_3}(0) = 5.5$ s, and $p_{s_4}(0) = 0$ s. Two possible synchronization trees T_1 and T_2 are shown in Fig. 2. T_1 is an optimal synchronization tree; it synchronizes s_1 and s_2 at time $t = 55$, s_3 and s_4 at time $t = 55$, and all streams at time $t = 150$. Hence, $cost(T_1) = (55+55+150+150) \times 1.5$ Mbits = 615 Mbits. T_2 synchronizes s_2 and s_3 at time $t = 40$, s_2 and s_4 at time $t = 95$, and all streams at $t = 150$. Hence, $cost(T_2) = (40 + 95 + 150 + 150) \times 1.5$ Mbits = 652.5 Mbits. Note that, if we were given a slightly different input, e.g., one where $p_{s_2}(0)$ and $p_{s_3}(0)$ are 8 s and 7 s, respectively, then T_2 would be the optimal tree, at a cost of 585 Mbits.

From the above example, it should be clear that the shape of an optimal synchronization tree depends on the initial values of the playback points of all streams. Thus, we have to consider how the remaining streams will be synchronized when making a choice of a pair of adjacent streams to be synchronized. Consider constructing the synchronization tree by choosing one pair of nodes (to merge) at a time, starting with the leaf nodes and working our way up. Then we would have $n-1$ possible pairs of streams from which to choose the first time, $n-2$ possible pairs of streams from which to choose the second time, and $n-i-1$ possible pairs of streams from which to choose the i-th time. Therefore, there are $(n-1)!$ non-unique binary synchronization trees [5]. Due to the inter-dependence between choices of pairs of streams to merge and a large number of possible ways to build a synchronization tree, it would be difficult or impossible to find an efficient algorithm which finds an optimal synchronization tree for all possible inputs. Thus, we concentrate on simple heuristic algorithms.

To simplify our discussion in the remainder of this section, we assume that $\Delta_+ = \Delta_- = \Delta$. For $2 \leq i \leq n$, let $life(s_i)$ be the period between time 0 and the time when the resources used by s_i are released due to merging with another stream. Let $life(s_1)$ be the synchronization time of the set of n streams, $\{s_1, \ldots, s_n\}$.

Definition 7. *The time cost of a synchronization tree T, corresponding to the set of n streams, $\{s_1, \ldots, s_n\}$, is defined as*

$$time(T) = \sum_{i=1}^{n} life(s_i).$$

Lemma 3. *Given a synchronization tree T,*

$$time(T) C_{min} \leq cost(T) \leq time(T) C_{max}$$

Proof. Since $\forall i (1 \leq i \leq n), C_{min} \leq C_{s_i}(t) \leq C_{max}$ for $t \in [0, P]$,

$$\sum_{i=1}^{n} life(s_i) \times C_{min} \leq \sum_{i=1}^{n} \int_0^P C_{s_i}(t)\, dt \leq \sum_{i=1}^{n} life(s_i) \times C_{max}$$

The result follows.

Theorem 1. *An upper bound on the time cost of a synchronization tree with the minimum time cost for the set of streams $\{s_1, \ldots, s_n\}$ is equal to $(\log_2(n)+1)P$, for all $n \geq 2$, where $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{2\Delta}$.*

Proof. We use mathematical induction to prove the theorem. Let $P(n)$ be "the minimum time cost of synchronizing a set of n streams $\{s_1, \ldots, s_n\}$ is less than or equal to $(\log_2(n) + 1)P$, where $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{2\Delta}$."

Basis step. For $n = 2$, it takes $P = \frac{p_{s_1}(0) - p_{s_2}(0)}{2\Delta}$ s to synchronize the streams. Hence, the time cost $= 2P = (\log_2(2) + 1)P$. Hence, $P(n)$ is true for $n = 2$.

Inductive step. Assume $P(n)$ is true for $n \leq k$ for some positive integer $k \geq 2$. For $n = k+1$, we divide the streams into two groups S_1 and S_2 such that $\forall s \in S_1$, $p_s(0) < \frac{p_{s_1}(0) + p_{s_n}(0)}{2}$ and $\forall s \in S_2, p_s(0) \geq \frac{p_{s_1}(0) + p_{s_n}(0)}{2}$.

[4] The resources used by s_b are released after it has been merged with s_a.

[5] The trees are not unique, because the same binary tree can be constructed in two or more different ways (i.e., corresponding to two or more different sequences of stream merges).

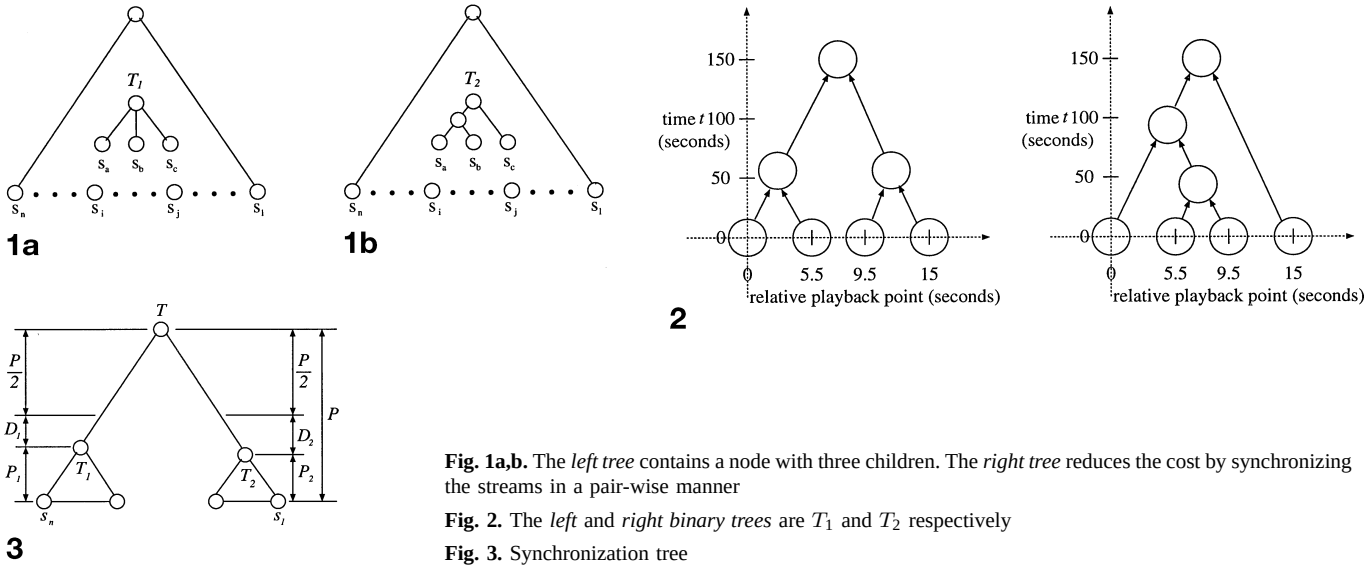


Fig. 1a,b. The *left tree* contains a node with three children. The *right tree* reduces the cost by synchronizing the streams in a pair-wise manner

Fig. 2. The *left* and *right binary trees* are T_1 and T_2 respectively

Fig. 3. Synchronization tree

We synchronize streams $s_1, \ldots, s_n$ using the following procedure: streams in S_1 are synchronized to become a stream denoted by s'; streams in S_2 are synchronized to become a stream denoted by s''; finally, s' and s'' are synchronized to become a single stream. The synchronization tree T for streams $s_1, \ldots, s_n$ is shown in Fig. 3. Let T_1 and T_2 be the synchronization trees for S_1 and S_2, respectively. Let n_1 and n_2 be $\mid S_1 \mid$ and $\mid S_2 \mid$, respectively. Let P_1 and P_2 be the synchronization time of S_1 and the synchronization time of S_2, respectively. Let D_1 and D_2 be $\frac{P}{2} - P_1$ and $\frac{P}{2} - P_2$, respectively. Since S_1 and S_2 must contain at least one stream, we have $1 \leq n_1 \leq k$ and $1 \leq n_2 \leq k$. From Fig. 3, we have

$$time(T) = 2 \times \tfrac{P}{2} + D_1 + D_2 + time(T_1) + time(T_2) .$$

By the inductive hypothesis,

$$time(T) \leq P + D_1 + D_2 + (\log_2(n_1) + 1)P_1 + (\log_2(n_2) + 1)P_2 .$$

Since $D_1 \log_2(n_1) \geq 0$ and $D_2 \log_2(n_2) \geq 0$, we have

$$time(T) \leq P + (P_1 + D_1)(\log_2(n_1) + 1) + (P_2 + D_2)(\log_2(n_2) + 1) .$$

By $\frac{P}{2} = D_1 + P_1 = D_2 + P_2$,

$$time(T) \leq P + \frac{P}{2}(\log_2(n_1) + 1) + \frac{P}{2}(\log_2(n_2) + 1)$$

$$\Rightarrow time(T) \leq P + \frac{P}{2}(\log_2(n_1 \times n_2) + 2) .$$

Since $n_1 + n_2 = k + 1$, $n_1 \times n_2$ has the maximum value when $n_1 = n_2 = \frac{k+1}{2}$. Therefore,

$$time(T) \leq P + \frac{P}{2}(\log_2(\frac{(k+1)^2}{2^2}) + 2)$$

$$\Rightarrow time(T) \leq (\log_2(k+1) + 1)P .$$

Thus, $P(n)$ is true for $n = k + 1$.

Corollary 1. *The cost of an optimal synchronization tree corresponding to streams $s_1, \ldots, s_n$ is not greater than $P(\log_2(n) + 1)C_{max}$, where $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{2\Delta}$.*

Proof. The result follows from Theorem 1 and Lemma 3.

Using the proof of Theorem 1, we can construct a recursive algorithm for finding a synchronization tree which has a time cost not greater than $(\log_2(n) + 1)P$. This recursive algorithm is used by the function BuildSyncTree, which is given below. (The function returns the root of the synchronization tree.)

```
function BuildSyncTree(set of streams S): tree node pointer
 var
     set of streams S1, S2;
     stream s', s'';
     tree node pointer t;
 begin
     t := new(tree node);
     if | S | > 1 then
         begin
             s' := first stream of S;
             s'' := last stream of S;
             S1 := {s ∈ S | ps(0) < (ps'(0) + ps''(0))/2};
             S2 := {s ∈ S | ps(0) ≥ (ps'(0) + ps''(0))/2};
             left child of t := BuildSyncTree(S1);
             right child of t := BuildSyncTree(S2);
         end
     else
         begin
             left child of t := NULL;
             right child of t := NULL;
             stream of t := the stream in S;
         end;
     return(t);
 end
```

Theorem 2. *Given a set of n streams, $\{s_1, \ldots, s_n\}$, algorithm BuildSyncTree requires $O(n \log(n))$ comparisons to find a synchronization tree corresponding to these n streams.*

Proof. Represent the set of streams by an array of n elements sorted in the descending order of the initial values

of playback points. In the worst case, each call to function BuildSyncTree splits the set of streams S into a set containing only a single stream and another set containing $\mid S \mid -1$ streams. Using binary search, we can locate the split point of S in $O(\log(\mid S \mid))$ comparisons. Therefore, in the worst case, the number of comparisons is $O(\sum_{i=2}^{n} \log(i)) = O(n \log(n))$.

Let K be the set of all possible sets of n streams with the same synchronization time P, where an element in K is characterized by the initial relative playback points of the n streams. In other words, since the time cost and the synchronization cost of n streams depend only on the relative playback points of those streams, we do not need to make a distinction between two sets of streams with identical initial relative playback points.

Theorem 3. *Let $S = \{s_1, \ldots, s_n\}$ be a set of n streams, where $S \in \mathsf{K}$ and S has the largest optimal time cost among the elements of K. Then the time cost of a synchronization tree with the minimum time cost, corresponding to S, is greater than or equal to $P \log_2(n)$, where $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{2\Delta}$ for all $n \geq 2$.*

Proof. Let $\mathsf{P}(n)$ be "the minimum time cost of synchronization of streams $s_1, \ldots, s_n$ is at least $n\tau \log_2(n)$ for some input where $\tau = \frac{p_{s_1}(0) - p_{s_n}(0)}{2(n-1)\Delta}$."
For $n = 2$, $\mathsf{P}(n)$ is true.
Assume $\mathsf{P}(n)$ is true for $n \leq k$ for some positive integer $k \geq 2$.
For $n = k + 1$, we choose $s_1, \ldots, s_n$ such that $\forall 0 \leq i \leq n-1$, $p_{s_i}(0) - p_{s_{i+1}}(0) = d$ for some positive real number $d > 0$.
Let T be a synchronization tree for $s_1, \ldots, s_n$. Let T_1 and T_2 be the subtrees of T. Let n_1 and n_2 be the number of streams that are covered by T_1 and T_2, respectively. We have

$$n_1 + n_2 = k + 1,$$
$$\tau = \frac{d}{2\Delta}.$$

By Lemma 1, the streams of T_1 and the streams of T_2 will be synchronized at time $t = (n_1 - 1)\tau$ and $t = (n_2 - 1)\tau$, respectively, and all streams will be synchronized at time $t = k\tau$. Therefore, the remaining lifetime of the stream at the root of T_1, after merging all streams in T_1, and the remaining lifetime of the stream at the root of T_2, after merging all streams in T_2, are $k\tau - (n_1 - 1)\tau$ and $k\tau - (n_2 - 1)\tau$, respectively. Thus,

$$\begin{aligned} time(T) &= k\tau - (n_1 - 1)\tau + k\tau - (n_2 - 1)\tau \\ &\quad + time(T_1) + time(T_2) \\ &\Rightarrow time(T) \geq (k+1)\tau + n_1 \log_2(n_1)\tau \\ &\quad + n_2 \log_2(n_2)\tau \text{ (by } n_1 + n_2 = k + 1 \\ &\quad \text{and inductive hypothesis)}. \end{aligned}$$

Let $F(x) = x \log_2(x) + (n-x) \log_2(n-x)$ for $1 \leq x \leq n-1$ where $n \geq 2$ is a constant.

$$\frac{dF(x)}{dx} = \log_2(x) - \log_2(n - x)$$

For $x = \frac{n}{2}$, $\frac{dF(x)}{dx} = 0$.
For $x > \frac{n}{2}$, $\frac{dF(x)}{dx} > 0$.
For $x < \frac{n}{2}$, $\frac{dF(x)}{dx} < 0$.
Hence, $F(x)$ is minimum when $x = \frac{n}{2}$ or $F(x) \geq n(\log_2(n) - 1)$.
Letting $x = n_1$ and $n = k + 1$, we have:

$$\begin{aligned} &time(T) \geq (k+1)\tau + F(\tfrac{k+1}{2})\tau \\ &\Rightarrow time(T) \geq (k+1)\tau + (k+1)(\log_2(k+1) - 1)\tau \\ &\Rightarrow time(T) \geq (k+1)\log_2(k+1)\tau \\ &\Rightarrow \mathsf{P}(n) \text{ is true for } n = k + 1. \end{aligned}$$

By mathematical induction, $\mathsf{P}(n)$ is true for all positive integers $n \geq 2$. Therefore, $time(T) \geq n \log_2(n)\tau \geq P \log_2(n)$. The result of the theorem follows as the time cost in the worst case is greater than or equal to the time cost in the case described above.

Corollary 2. *The cost of an optimal synchronization tree corresponding to the set of streams $\{s_1, \ldots, s_n\}$ is greater than or equal to $P \log_2(n) C_{min}$ for some input where $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{2\Delta}$.*

Proof. By Theorem 3, there exists some input such that if T' is the synchronization tree with the minimum time cost, then

$$time(T') \geq P \log_2(n).$$

Let T be an optimal synchronization tree. We have

$$\begin{aligned} &P \log_2(n) \leq time(T') \leq time(T) \\ &\Rightarrow P \log_2(n) C_{min} \leq time(T) C_{min} \\ &\Rightarrow P \log_2(n) C_{min} \leq time(T) C_{min} \\ &\leq cost(T) \text{ (by Lemma 3)}. \end{aligned}$$

Recall that K is the set of all possible sets of n streams with the same synchronization time P, where an element in K is characterized by the initial relative playback points of the n streams. Then we have the following corollary.

Corollary 3. *Let $S \in \mathsf{K}$ be the set of n streams with the largest optimal synchronization cost among the elements of K; then S satisfies the following condition:*

$$\begin{aligned} P log_2(n) C_{min} &\leq \textit{the optimal synchronization cost of } S \\ &\leq P(log_2(n) + 1) C_{max}. \end{aligned}$$

Proof. The result follows from Corollary 1 and Corollary 2.

Note that the upper and lower bounds diverge logarithmically, so the bounds on the worst case cost are tight.

4 Dynamic stream-merging algorithms

In a VOD system, a stream is initiated for an object when a new request for that object arrives, and it is removed from the system when the stream reaches the end of the object's display. Therefore, we have to consider the dynamic stream-merging problem in order to optimize the I/O demand reduction resulting from the stream-merging approach. Thus, in this section, we describe several dynamic stream-merging algorithms. We consider a class of dynamic stream-merging algorithms which make speed adjustments when one of the following two types of events occurs: *arrival* or *merge*.

An *arrival* event corresponds to an initiation of a new I/O stream. A *merge* event corresponds to a merge of two I/O streams.

The dynamic stream-merging problem is much more complex than the static stream-merging problem. When an event occurs, an algorithm which minimizes the I/O demand must consider many factors, such as, the request arrival process and the current playback points of the existing streams. Complex stochastic modeling and optimization would be required to find the best way to merge the streams corresponding to requests for the same object.

However, we can find a good heuristic for solving the dynamic stream-merging problem by making use of an algorithm constructed for solving the static stream-merging problem. More specifically, we can break down a dynamic stream-merging problem into several smaller static stream-merging problems. The basic idea is to partition the streams of the same video object into disjoint sets such that the streams in each set can be merged into a single stream before any of the streams in this set terminate (due to reaching the end of an object's display). One possible partitioning approach is to group the streams according to their arrival times such that all the streams in the same group have arrived within a time period, called the *catch-up window*. We use the term 'window-based' stream-merging algorithm to refer to a dynamic stream-merging algorithm that partitions streams into disjoint sets using a catch-up window. In the remainder of this section, we present several dynamic merging algorithms, whose performance is analyzed and compared in Sects. 5 and 6, respectively.

4.1 Baseline algorithm

This is the normal situation. When a request arrives, there is no attempt to adjust the display rates, i.e., all requests are assigned the normal display rate, and there are no merging events in the system.

4.2 Equal-Split algorithm

This algorithm partitions I/O streams into groups using a catch-up window. For each group of streams, algorithm BuildSyncTree is used to find a way to merge the streams in the group. The streams are partitioned into groups such that the period between the arrivals of any two streams in the same group is bounded by W seconds, where W is the size of the catch-up window. (If the offline approach to speed alteration is used (see Sect. 1), then the value of W should be chosen to strike a balance between the additional disk storage required and the resulting I/O demand reduction. Section 6.2 discusses this tradeoff in more detail, and Fig. 9 illustrates it.) The partitioning of I/O streams is done as follows. If a stream s arrives at time t and there does not exist a group of streams such that the arrival time of the leading stream (the group leader) is less than $t-W$, then s becomes the leader of a new group. All streams which arrive in the period between time t and $t+W$ belong to this new group. The term *current* group refers to the group of streams that has been created most recently. The details of the algorithm Equal-Split are given below.

```
procedure Equal-Split(Event event, float W)
  var
    set of streams G;
  begin
    if (event is an arrival of stream s at time t) then
      if (there does not exist a group of streams such that the leading
      stream of the group has arrived after time t − W) then
        begin
          create a new group and put s into the new group;
          speed of s := S_min
        end
      else
        begin
          G := the set of streams in the current group;
          put s into G;
          call BuildSyncTree(G) to find a synchronization tree
          for the streams in G;
        end
    else if (event is a merge of streams s_i and s_j) then
      begin
        discard s_j;
        if (s_i is the only stream in its group) then
          speed of s_i := S_n
        else if (s_i is going to merge with a slower stream according to
              an already constructed synchronization tree) then
          speed of s_i := S_min
        else
          speed of s_i := S_max;
      end
  end
```

The size of a catch-up window, i.e., W, is a tuning parameter of the window-based algorithms. Note that there are limitations on the actual value of W; thus we proceed by deriving the maximum possible value of W. The time required to merge the leading stream and the trailing stream in some group must be less than the remaining lifetime of the leading stream. Note that the leading stream is at most $(1-\Delta_-)W$ seconds ahead of the trailing stream, because the leading stream moves at a speed of $(1-\Delta_-)S_n$ until all streams have been merged. Hence, if L is the normal playback time of a video object, then

$$\frac{(1-\Delta_-)W}{\Delta_+ + \Delta_-} \leq \frac{L-(1-\Delta_-)W}{1-\Delta_-} ,$$

or

$$W \leq \frac{(\Delta_+ + \Delta_-)L}{(1+\Delta_+)(1-\Delta_-)} .$$

4.3 Brute-force algorithm

This algorithm is the same as the Equal-Split algorithm, except that it finds an optimal synchronization tree for the streams in each group by evaluating all the possible synchronization trees. As mentioned in Sect. 3, given a group of n streams, we can construct $(n-1)!$ (corresponding) non-unique binary synchronization trees.

4.4 Offline brute-force algorithm

This algorithm is the same as the brute-force algorithm, except that it knows the arrival times of requests a priori, and hence it performs better than the online brute-force algorithm and the Equal-Split algorithm.

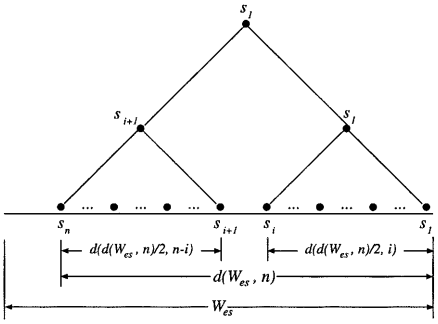


Fig. 4. Synchronization tree for n streams under algorithm Equal-Split

5 Performance analysis

In this section, we present performance analysis of the stream-merging algorithms described in Sect. 4, using the mean total I/O demand, on a storage server, as the measure of performance. We define the following notation to be used in the derivations of this section. All analysis is done with respect to a particular video object. Finally, we assume that the request arrival process is Poisson.

- L = normal playback time of a video object
- λ = mean request arrival rate
- t_a = random variable representing the time between successive I/O stream initiations
- W_a = size of a catch-up window for algorithm a (note that we have already defined W in Sect. 4)
- BW_a = mean total I/O bandwidth demand under algorithm a (bits/s)

Since the request arrival process is Poisson with rate λ, the probability density function of t_a is

$$f_{t_a}(x) = \lambda e^{-\lambda x} \quad \text{for } x \geq 0 .$$

5.1 Analysis of the baseline algorithm

There is no merging of streams in the baseline algorithm. Thus, the mean total bandwidth demand under this algorithm is the product of the average number of streams that are active simultaneously and the normal display bandwidth. Hence, the expected I/O demand is

$$BW_b = \lambda L C_n .$$

5.2 Analysis of the Equal-Split algorithm

The behavior of the Equal-Split algorithm is such that each group of streams is statistically identical. We can therefore analyze the mean I/O demand for one such group of streams (and then use the results to compute the mean I/O demand for the system). Consider Fig. 4, which depicts a system with n streams.

Let $d(w,k)$ and $B(w,k)$ be the mean time between arrivals of s_i and s_{i+k-1} and the mean I/O cost (in bits) of merging streams $s_i, \ldots, s_{i+k-1}$, respectively, given that (a) exactly k streams have arrived in a period of w time units, (b) stream s_i has arrived at the beginning of the period, and (c) streams $s_{i+1}, \ldots, s_{i+k-1}$ have arrived during this period. Given that k streams have arrived in a catch-up window of length w, $d(w,k)$ is the mean time between the arrivals of the leading and the trailing streams in this set of k streams, and $B(w,k)$ is the mean I/O cost (in bits) of merging these k streams. Hence, $d(W_{es}, n)$ and $B(W_{es}, n)$ are the mean time between arrivals of s_1 and s_n and the mean I/O cost of merging streams $s_1, \ldots, s_n$, respectively. $d(w,k)$ can be expressed as

$$d(w,k) = \frac{k-1}{k} w \quad \text{if } k > 1 .$$

At this point we can derive $B(w,k)$. When $k = 2$, two streams, s_i and s_{i+1}, are merged into one stream. Since s_i has a display rate of $S_{min} = (1 - \Delta_-)S_n$ before the merge and the time between arrivals of s_i and s_{i+1} is $d(w,2)$, $p_{s_i}(t) - p_{s_{i+1}}(t) = (1 - \Delta_-)d(w,2)$ when the merge begins at time t. Then the cost of merging s_i and s_{i+1}, $B(w,2)$, is equal to $d(w,2)(\frac{(1+\Delta_+)C_{min}}{\Delta_+ + \Delta_-} + \frac{(1-\Delta_-)C_{max}}{\Delta_+ + \Delta_-})$. When $k > 2$, let $i+1$ be the number of streams covered by the left subtree of the root of the synchronization tree. The probability that the left subtree covers $i + 1$ streams is given by the following binomial distribution,

$$2^{-(k-2)} \frac{(k-2)!}{(k-2-i)!\,i!} , \quad \text{where} \quad 0 \leq i \leq k-2 .$$

By taking the products of this probability and the corresponding I/O cost and then summing over all possible cases, we can derive the value of $B(w,k)$, which can be expressed as

$$B(w,k) = \begin{cases} d(w,2)(U_1 + U_2) & \text{if } k = 2, \\ 2^{-(k-2)}\{\sum_{i=1}^{k-3} C_i^{k-2}[B(\frac{d(w,k)}{2}, i+1) & \\ +B(\frac{d(w,k)}{2}, k-i-1)+ & \\ (d(w,k) - d(\frac{d(w,k)}{2}, i+1))U_1 & \\ +(d(w,k) - d(\frac{d(w,k)}{2}, k-i-1))U_2]+ & \\ (2d(w,k) - d(\frac{d(w,k)}{2}, k-1))(U_1 + U_2) & \\ +2B(\frac{d(w,k)}{2}, k-1)\} & \text{if } k > 2, \end{cases}$$

where $U_1 = \frac{(1+\Delta_+)C_{min}}{\Delta_+ + \Delta_-}$, $U_2 = \frac{(1-\Delta_-)C_{max}}{\Delta_+ + \Delta_-}$, and $C_i^k = \frac{k!}{(k-i)!\,i!}$. Since, $B(w,k) = B(1,k)w$, we can express $B(w,k)$ as

$$B(w,k) = wB'(k) ,$$

where

$$B'(k) = \begin{cases} \frac{U_1+U_2}{2} & \text{if } k = 2. \\ \frac{k-1}{k2^{k-2}}\{\frac{1}{2}\sum_{i=1}^{k-3} C_i^{k-2}[B'(i+1) & \\ +B'(k-1-i) + \frac{i+2}{i+1}U_1 & \\ +\frac{k-i}{k-i-1}U_2]+ & \\ \frac{3k-2}{2(k-1)}(U_1 + U_2) + B'(k-1)\} & \text{if } k > 2. \end{cases}$$

Since $B'(k)$ can be solved recursively, we can compute $B(w,k)$. Let BW_{es}^n be the mean total I/O cost for the n streams in Fig. 4. Then, BW_{es}^n can be expressed as follows.

$$BW_{es}^n = \begin{cases} W_{es}C_{min} + [L - (1 - \Delta_-)W_{es}]C_n & \text{if } n = 1 \\ B(W_{es}, n) + [L - \frac{(1-\Delta_-)d(W_{es},n)}{\Delta_+ + \Delta_-}]C_n & \text{if } n \geq 2 \end{cases}$$

Since a new group of streams is initiated when a request arrives after the end of the catch-up window of the previous group and is terminated when the new catch-up window expires, the mean time between initiations of two consecutive groups is $W_{es}+1/\lambda$. The expected I/O demand for algorithm Equal-Split is the product of the initiation rate of groups of streams and the mean total I/O cost of a group; this expected I/O demand can be expressed as

$$BW_{es} = \frac{1}{W_{es}+1/\lambda}\sum_{n=0}^{\infty}\frac{(\lambda W_{es})^n e^{-\lambda W_{es}}}{n!}BW_{es}^{n+1}\ .$$

5.3 Validation of analytic result

In conclusion of this section, we validate our (approximate) analysis of algorithm Equal-Split by comparing it with results obtained through simulation. The performance measure which we consider is the percentage reduction of (or improvement in) the average bandwidth requirement of a system, due to algorithm Equal-Split, as compared to the baseline algorithm. Figure 5 shows the percentage bandwidth improvement for several different catch-up windows. The curves in Fig. 5 indicate that the analytic results match the simulation results closely (for most cases we have considered). Therefore, the analytic results should be sufficient for the performance evaluation of the algorithm under different catch-up windows and arrival rates. Note however, that the difference between analytic results and simulation results becomes larger when the length of the catch-up window is maximum (in this case, when W_a = 12 min). This is due to the fact that, in the analysis of Sect. 5.2, we assumed that the arrival times of all streams in the same group are known a priori, and thus the synchronization tree can be constructed according to the arrival times of all the streams in the group[6]. The error caused by this assumption increases as the number of requests in the group increases.

6 Performance of merging algorithms

In this section, we compare the performance of the Equal-Split algorithm, the brute-force algorithm, the offline brute-force algorithm, and two algorithms proposed in [5]. In [5], the authors proposed the odd-even, simple, and greedy algorithms. The odd-even algorithm attempts to reduce I/O demand by at most 50%. The basic idea behind the odd-even algorithm is to pair up and merge two consecutive streams whenever possible. The simple algorithm merges streams in a group by slowing down the leading stream of the group and accelerating all streams trailing the leader. In the greedy algorithm, adjacent streams are merged until no further merging of streams is possible. (It is basically a recursive application of the odd-even algorithm.) We have evaluated the performance of the merging algorithms by computer simulation, because we do not have the analytic results for the brute-force and the offline brute-force algorithms [7].

Table 1. Values of parameters used in simulation

Parameter	Value
L	120 min
S_{min}	28.5 frames/s
S_n	30 frames/s
S_{max}	31.5 frames/s
Δ_+	0.05
Δ_-	0.05
C_{min}	1.425 Mbits/s
C_n	1.5 Mbits/s
C_{max}	1.575 Mbits/s

The maximum possible catch-up window (for the cases discussed in this section) is W_a = 12 min, when the normal display time of a video object is 120 min and $\Delta_+ = \Delta_- = 0.05$. We have evaluated the performance of the algorithms for the maximum catch-up window (W_a = 12 min), a catch-up window of medium length (W_a = 6 min), and a relatively short catch-up window (W_a = 2 min). The performance of each stream-merging algorithm was measured as the percentage reduction of (or improvement in) the average bandwidth requirement of a system, as compared to the baseline algorithm. The values of the parameters used in the simulations are given in Table 1. The stream arrival process was modeled as a Poisson process. The results are presented with 95% confidence intervals where the length of each confidence interval is bounded by 0.1%. Figures 6-8 show the performance results for the maximum possible catch-up window W_a = 12 min, W_a = 6 min, and W_a = 2 min, respectively [8].

Because of the extremely long computation time ($O((n-1)!)$) of algorithm Brute-force, the simulation results for this algorithm could not be obtained for high arrival rates, at least not within 480 h of CPU time when W_{es} was between 6 min and 12 min.

The online brute-force, Equal-Split, and greedy algorithms achieve a high I/O demand reduction at high arrival rates. The brute-force and Equal-Split algorithms can result in reductions in I/O demand of more than 80% when the mean inter-arrival time is 0.5 min and of more than 30% at relatively low arrival rates [9]. The results in Figs. 6-8 show that algorithm Equal-Split outperforms the odd-even and greedy algorithms in all cases. With a catch-up window of 12 min and the mean inter-arrival times of 0.5 to 10 min, algorithm Equal-Split outperforms the odd-even algorithm by 28% to 70%, and it outperforms the greedy algorithm by 5–20%.

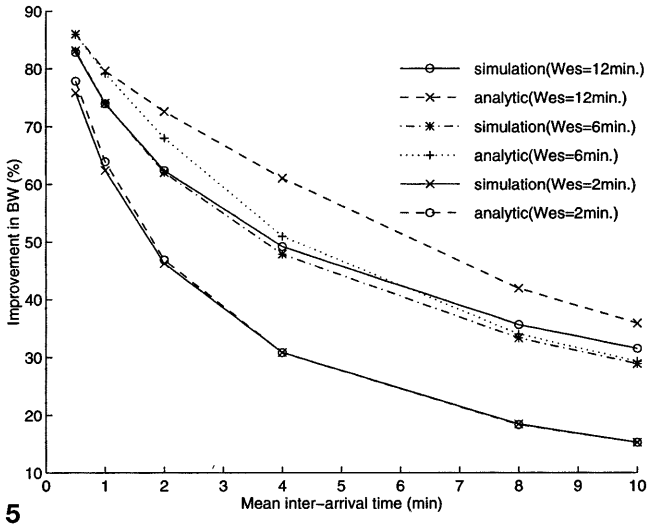
The (online) brute-force algorithm performs only slightly better than the Equal-Split algorithm in all cases for which

[6] That is, assuming that arrival times of all streams in the same group are known a priori results in the assumption that merging decisions are never "reversed". This is not the case in algorithm Equal-Split, where it is possible to "waste" some effort, because some merging decisions may be "reversed" as a result of new arrivals.

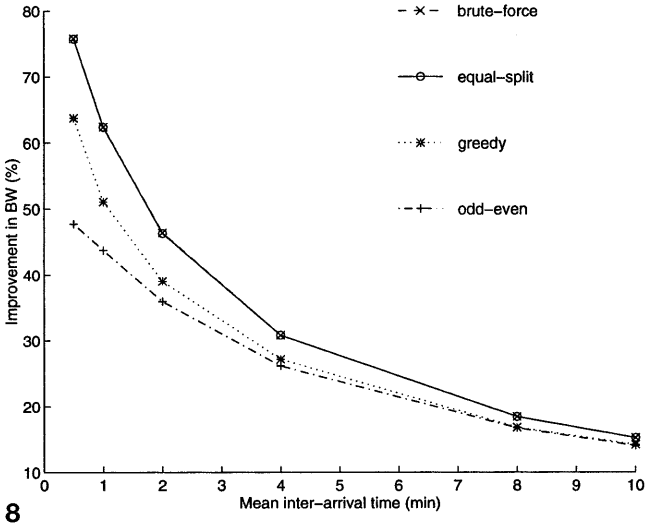
[7] It appears, from the graphs reported in [5], that the simple algorithm performs worse than the other two algorithms presented in the same paper, at least for the Poisson arrival process. Hence, we do not include the simple algorithm in our comparison.

[8] We have modified the original odd-even and greedy algorithms in [5] such that W_a is a tuning parameter of the algorithms rather than something that is computed by the algorithm itself, as was defined in [5].

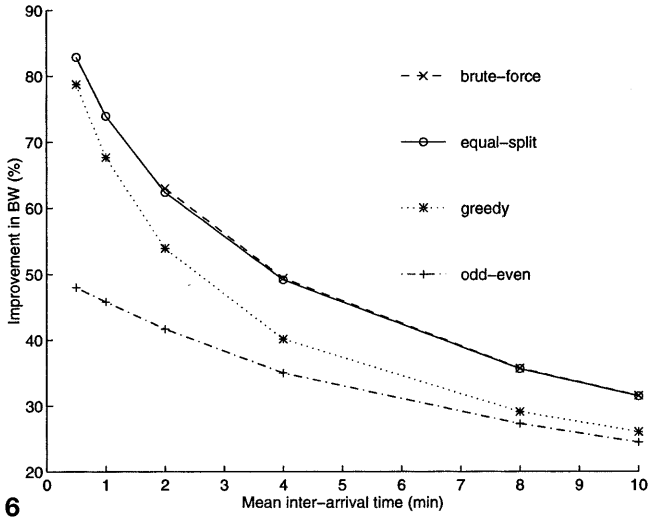
[9] We refer to the cases for which it was possible to compute performance results.



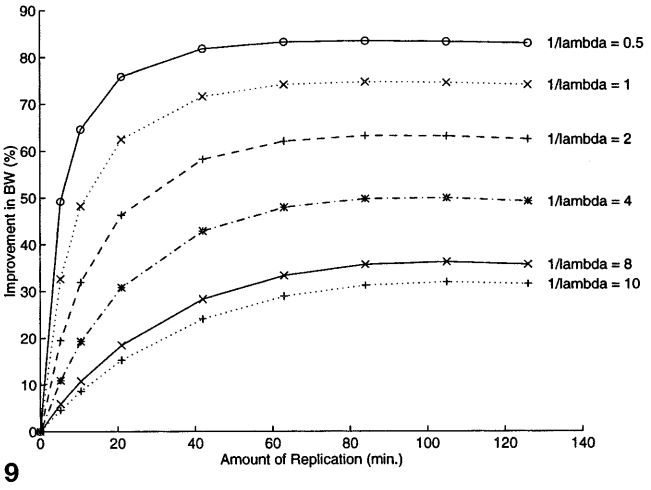
5



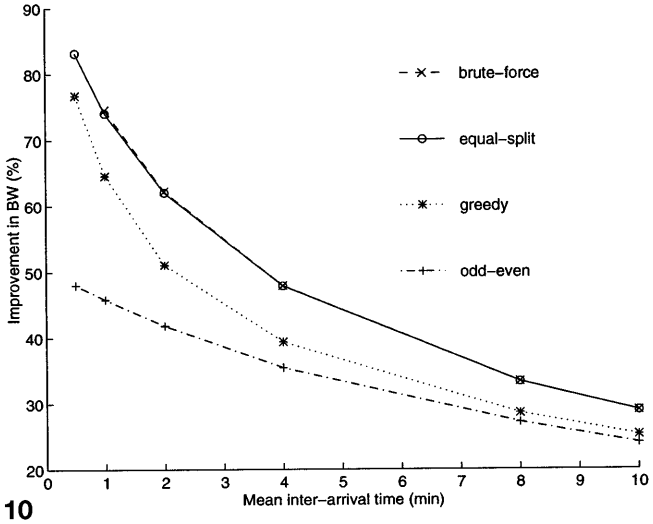
8



6



9



10

Fig. 5. Validation of analysis of algorithm Equal-Split

Fig. 6. Performance of merging algorithms when W_a = 12 min

Fig. 7. Performance of merging algorithms when W_a = 6 min

Fig. 8. Performance of merging algorithms when W_a = 2 min

Fig. 9. Algorithm Equal-Split

Table 2. Bandwidth improvement when W_a = 12 min

Mean inter-arrival time (min)	Bandwidth improvement (%)		
	Offline brute-force	Brute-force	Equal-Split
0.5	NA	NA	82.88
1.0	NA	NA	73.96
2.0	NA	63.00	62.37
4.0	52.59	49.42	49.17
8.0	38.04	35.74	35.66
10.0	33.58	31.58	31.53

Table 3. Bandwidth improvement when W_a = 6 min

Mean inter-arrival time (min)	Bandwidth improvement (%)		
	Offline brute-force	Brute-force	Equal-Split
0.5	NA	NA	83.16
1.0	NA	74.57	74.06
2.0	63.82	62.19	62.00
4.0	49.05	47.91	47.87
8.0	34.05	33.34	33.32
10.0	29.50	28.90	28.90

it was possible to compute performance results. Indeed, the performance difference of the two algorithms is under 1%.

6.1 Offline brute-force algorithm

We evaluated the performance of the offline brute-force algorithm for several different catch-up windows. Tables 2–4 show the percentage bandwidth improvement of the Equal-Split, brute-force, and offline brute-force algorithms for 12-, 6-, and 2-min catch-up windows, respectively. Note that we can only obtain the performance of the (online) brute-force and the offline brute-force algorithms for low-to-medium arrival rates because of the extremely long computation time of the algorithms. We ran the simulations on a Sun Sparc 2000 workstation with 20 processors. We parallelized the procedure for searching for the optimal synchronization tree to harness the power of 20 processors. Each table entry marked with 'NA' corresponds to a case in which a simulation run could not be completed within 480 h of CPU time.

Since the offline brute-force algorithm has perfect knowledge of arrival times of requests, it performs better than the Equal-Split algorithm and the online brute-force algorithm[10]. However, for low-to-medium arrival rates, it performs only slightly better than its online counterpart, brute-force algorithm, or the Equal-Split algorithm. Also, the online (or offline) brute-force algorithm can perform slightly better than the Equal-Split algorithm when the window length is 2 min and the mean inter-arrival time is 0.5 min.

6.2 Limited merging

The alteration of display rates can be implemented by storing replicas of a video object for display rates $(1 - \Delta_-)S_n$ and $(1 + \Delta_+)S_n$. If replication of data is used to perform display rate alteration, then we need to consider the amount of additional disk space that would be necessary to store the replicated data. Note that replicated data for speed-up and slow-down is required while the streams in a group are being merged and is not required after the streams in a group have been merged into a single stream. A smaller amount of additional storage is required for a smaller catch-up window, because the amount of replicated data grows with the time required to merge all streams in a group and hence grows with the length of the catch-up window. On the other hand, the larger the catch-up window, the greater the opportunity for I/O bandwidth demand reduction. Thus, there is a tradeoff between the increase in storage requirements and the reduction in I/O bandwidth demand.

[10] Note that the offline brute-force does not necessarily minimize the I/O cost for the dynamic stream-merging problem.

Table 4. Bandwidth improvement when W_a = 2 min

Mean inter-arrival time (min)	Bandwidth improvement (%)		
	Offline brute-force	Brute-force	Equal-Split
0.5	76.46	75.84	75.82
1.0	62.87	62.43	62.40
2.0	46.59	46.42	46.29
4.0	31.00	30.82	30.82
8.0	18.56	18.46	18.46
10.0	15.35	15.27	15.27

We have investigated the performance of algorithm Equal-Split under different lengths of catch-up windows. The results are shown in Fig. 9. Given fairly small mean inter-arrival times, most of the reduction in I/O demand can be achieved using fairly small catch-up windows. This implies that most of the reduction in I/O demand can be achieved with only a small amount of storage overhead. For example, when the mean inter-arrival time is 0.5 min and the catch-up window is 1 min long, the reduction in I/O demand is 64.58%, as compared to 82.88% when using the maximum possible catch-up window. However, the corresponding increase in disk storage (for 120 min video) would be $\approx$ 235 MB or 17% of the size of a video object for the 1-min catch-up window and $\approx$ 2.7 GB or 200% of the size of a video object for the maximum possible catch-up window.

7 Conclusions

In summary, we have formalized the static stream-merging problem, which minimizes the cost of merging a set of n streams, corresponding to requests for the same object, into a single stream, given that it is possible to merge all n streams. Our cost model is general and can be applied to many different architectures of VOD systems. The time required to merge a set of streams depends only on the initial difference of the playback points of the trailing and the leading streams in that set (Lemma 1), the maximum fraction of the display rate by which a stream can be sped up, and the maximum fraction of the display rate by which a stream can be slowed down. We have also proposed an efficient heuristic algorithm (BuildSyncTree), which requires $O(n \log(n))$ comparisons and finds a stream-merging order with an I/O cost not higher than $P(\log_2(n) + 1)C_{max}$, for $n > 1$, where $P = \frac{p_{s_1}(0) - p_{s_n}(0)}{2\Delta}$ is the time required to merge all n streams in a set and C_{max} is the bandwidth requirement of the maximum display rate. Based on algorithm BuildSyncTree, we have proposed a heuristic algorithm, Equal-Split, for solving the dynamic stream-merging problem. Although the offline brute-force algorithm (also introduced in the context of

dynamic stream-merging) has perfect knowledge of request arrival times and can find an optimal way to merge each group of streams of the same object which have arrived in a catch-up window, simulation results indicate that the performance of the Equal-Split algorithm is very close to that of the offline brute-force algorithm[11].

Moreover, we have shown that the optimum cost of merging n streams, in the static merging problem, is not greater than $P(\log_2(n)+1)C_{max}$ (Corollary 1). Algorithm Equal-Split partitions the streams of the same video object into disjoint sets using a catch-up window of length W seconds and calls Algorithm BuildSyncTree to merge the streams in each set. The playback points of the leading stream and the trailing stream of a set of streams differ by at most $(1-\Delta)W$ when the trailing stream arrives to the system [12]. Hence, the total I/O cost of a set of n streams[13] is not greater than $P(\log_2(n)+1)C_{max}+(L-P)C_n+nWC_{max}$, where $P = \frac{(1-\Delta)W}{2\Delta} \leq L$ and L is the normal display time of the video. Therefore, the total I/O cost of a set of n streams is not greater than $(P\log_2(n)+L)C_{max}+nWC_{max} \leq (\log_2(n)+1)LC_{max}+\frac{2\Delta}{(1+\Delta)(1-\Delta)}nLC_{max}$. Thus, the total I/O cost of the set of n streams is reduced by at least a factor of $\frac{nC_n}{\{\log_2(n)+1+\frac{2\Delta}{(1+\Delta)(1-\Delta)}n\}C_{max}}$. The savings in I/O bandwidth are substantial even for a small number of streams.

The cost/benefit tradeoff considered in this paper is the balance between the reduction in I/O bandwidth demand and the amount of storage overhead required for each video, i.e., we should only apply the stream-merging approach to a request for a given video when the benefit due to the I/O bandwidth demand reduction is greater than the cost of the storage overhead. Note that, in our algorithms, we have considered merging of requests for a single object, without considering how that affects requests for other objects. i.e., we have treated the I/O bandwidth resource allocated for servicing requests for one object independently of servicing requests for other objects. In practice, these are not independent; the I/O bandwidth demand of a pair of streams in the process of being merged could be larger than in the case where no attempts at merging are made, depending on the relative values of C_{min} and C_{max}. Therefore, merging of two streams corresponding to requests for one object could deprive requests for another object of the necessary bandwidth. However, we expect this effect to be quite small, since the deviation from the normal display rate is only a few percent, and thus we expect the difference between C_{min} and C_{max} to be small.

Further work is required to either prove that there is an efficient algorithm for finding an optimal way to merge streams, both in the case of the static and the dynamic stream-merging problems, or prove that the problem of finding an optimal way to merge streams is in NP.

[11] Note that the offline brute-force does not necessarily minimize the I/O cost for the dynamic stream-merging problem.

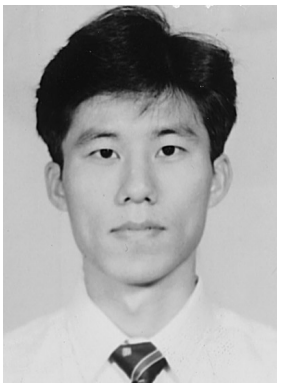
[12] The display rate of the leading stream is S_{min} before all the streams in the set are merged.

[13] Recall that the definition of the static merging problem guarantees that it is possible to merge all n streams into a single stream. When the solution to the static merging problem, Equal-Split, is used to construct a solution to the dynamic merging algorithm, the actual value of n depends on the inter-arrival distribution as well as on the size of the catch-up window, W.

Acknowledgements. The research done by S.W. Lau and J.C.S. Lui was supported in part by the UGC Earmarked Grant and the Direct Research Grant.

References

1. Beakley GW (1991) Channel Coding for Digital HDTV Terrestrial Broadcasting. In: IEEE Trans Broadcasting, Vol. 37, No. 4, pp. 137–140
2. Berson S, Ghandeharizadeh S, Muntz RR, Ju X (1994) Staggered Stripping in Multimedia Information Systems. In: ACM SIGMOD Conference, pp. 79–90, Minneapolis, Minnesota
3. Dan A, Shahabuddin P, Sitaram D, Towsley D (1994) Channel Allocation under Batching and VCR Control in Movie-on-Demand Servers. Technical Report, IBM Research Report
4. Kamath M, Towsley D, Ramamritham K (1994) Buffer Management for Continuous Media Sharing in Multimedia Database System. Technical Report TR:4-11, University of Massachusetts
5. Golubchik L, Lui JCS, Muntz RR (1995) Reducing I/O demands in Video-On-Demand Storage Servers. In: Proceedings of the ACM SIGMETRICS/PERFORMANCE '95 Conference, Ottawa, Canada, May, pp. 25–36
6. Le Gall D (1991) MPEG: A Video Compression Standard for Multimedia Applications. Commun ACM, April, pp. 46–58
7. Lau SW, Lui JCS, Wong PC (1995) A Cost-effective Near-line Storage Server for Multimedia System. In: Proceedings of the 11^{th} International Conference on Data Engineering, pp. 449–456, Taipei, Taiwan, March
8. Lau SW, Lui JCS (1995) A Novel Video-On-Demand Storage Architecture for Supporting Constant Frame Rate with Variable Bit Rate Retrieval. In: Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video, pp. 316–326, New Hampshire, April
9. Lau SW, Lui JCS (1996) Scheduling and Replacement Policies for a Hierarchical Multimedia Storage Server. In: Proceedings of Multimedia Japan 96, International Symposium on Multimedia Systems. pp. 68–75, Yokohama, Japan, March 18–22
10. Ozden B, Biliris A, Rastogi R, Silberschaz A (1994) A Low-cost Storage Server for Movie on Demand Databases. In: Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile, Sept., pp. 594–605
11. Rotem D, Zhao J (1995) Buffer Management for Video Database Systems. In: Proceedings of the 11^{th} International Conference on Data Engineering, pp. 439–448, Taipei, Taiwan, March
12. Vin HM, Rangan PV (1993) Designing a Multi-User HDTV Storage Server. IEEE J Select Areas Commun 11, No. 1, pp. 153–164



SIU-WAH LAU received his B.Sc. and M.Phil. in Computer Science from the University of Hong Kong in 1988 and 1991, respectively. He received his M.Sc. in Computer Science from UCLA in 1993. Currently, he is a PhD candidate in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include distributed multimedia systems and computer networks.

JOHN CHI-SHING LUI received his PhD in Computer Science from UCLA in 1991. He then joined a team in IBM Almaden/San Jose in the research and development of a parallel I/O architecture project. In 1993, he joined the Department of Computer Science and Engineering in the Chinese University of Hong Kong. His research interests are in parallel and distributed systems design, distributed multimedia systems, parallel I/O architectures, communication networks, mobile computing and performance evaluation theory.

LEANA GOLUBCHIK is an Assistant Professor in the Department of Computer Science, University of Maryland at College Park; from Fall of 1995 until Summer of 1997, she was an Assistant Professor in the Department of Computer Science at Columbia University. Her current research interests are in multimedia information systems, high-performance I/O, and computer systems modeling and performance evaluation. Dr. Golubchik received her BS in Computer Science and Engineering in 1989, her MS in Computer Science in 1992, and her PhD in Computer Science in 1995, all from the University of California, Los Angeles (UCLA). She is a member of Tau Beta Pi, the Association for Computing Machinery, and the IEEE.