# Resource Scheduling in Enhanced Pay-Per-View Continuous Media Databases

Minos N. Garofalakis*
University of Wisconsin–Madison
*minos@cs.wisc.edu*

Banu Özden
Bell Laboratories
*ozden@research.bell-labs.com*

Avi Silberschatz
Bell Laboratories
*avi@research.bell-labs.com*

## Abstract

The enhanced pay-per-view (EPPV) model for providing continuous-media-on-demand (CMOD) services associates with each continuous media clip a display frequency that depends on the clip's popularity. The aim is to increase the number of clients that can be serviced concurrently beyond the capacity limitations of available resources, while guaranteeing a constraint on the response time. This is achieved by sharing periodic continuous media streams among multiple clients. In this paper, we provide a comprehensive study of the resource scheduling problems associated with supporting EPPV for continuous media clips with (possibly) different display rates, frequencies, and lengths. Our main objective is to maximize the amount of disk bandwidth that is effectively scheduled under the given data layout and storage constraints. This formulation gives rise to $\mathcal{NP}$-hard combinatorial optimization problems that fall within the realm of hard real-time scheduling theory. Given the intractability of the problems, we propose novel heuristic solutions with polynomial-time complexity. Preliminary results from an experimental evaluation of the proposed schemes are also presented.

## 1 Introduction

With all the euphoria surrounding the potential benefits of the coming multimedia revolution, database researchers are faced with challenges that are pushing the current hardware and software technology to its limits. The fundamental problem in developing high-performance multimedia servers is that images, audio, and other similar forms of data differ from numeric data and text in their characteristics, and hence require different techniques for their organization and management. The most critical of these characteristics is that digital audio and video *streams* consist of a sequence of media quanta which convey meaning only when presented continuously in time. Hence, in contrast to traditional storage managers, a multimedia server needs to ensure that the retrieval and storage of such *continuous media* (CM) streams proceed at their pre-specified real-time rates. Given the limited amount of resources (e.g., memory, disk bandwidth, and disk storage), it is a challenging problem to design effective resource management algorithms that can provide on-demand support for a large number of concurrent continuous media clients. The service models supported by such Continuous-Media-On-Demand (CMOD) servers can be classified into two broad categories: *random access* and *enhanced pay-per-view*.

- **The Random Access service model** places resource reservations to allocate independent physical channels to each individual client. Under Random Access service, the maximum number of concurrent clients that can be supported is limited by the available resources. Such levels of concurrency may not be sufficient to provide cost-effective services in environments like Movies-On-Demand, where the client population far exceeds the maximum number of concurrent streams [10].
- **The Enhanced Pay-Per-View (EPPV) service model** aims to increase the number of clients that can be serviced concurrently beyond the limitations of available resources while guaranteeing a constraint on the response time. This is achieved by assigning with each CM clip (referred to as a clip in the remainder of the paper) a *display frequency*, typically determined by the clip's popularity, and sharing streams among multiple clients. Under the EPPV service model, the response time for transmission of a clip to a client is bounded by the reciprocal of the clip's display frequency (i.e., its *retrieval period*). From the client's perspective, perhaps the most attractive feature of EPPV service is that the client can be informed exactly when the transmission will start. Thus, even when resources are scarce the EPPV service model can guarantee *predictable* response times for all incoming requests. From the service provider's perspective, the most attractive feature of EPPV service is that the number of concurrent clients is not upper-bounded by resource availability.

The EPPV model for continuous media services is becoming more and more popular with the telecom, cable, broadcast, and content companies since it offers the potential to provide scalable, cost-effective CMOD offerings [16]. Realizing this potential, however, requires schemes for effectively scheduling the available disk bandwidth and storage capacity so that high levels of concurrency and system utilization can be sustained. Two phenomena make this a challenging problem — the periodic nature of EPPV service and the relatively high latencies of magnetic disk storage. The periodicity of clip retrievals in EPPV servers generates a host of difficult periodic task scheduling problems that fall within the realm of hard real-time scheduling theory [9]. The high disk latencies complicate effective utilization of disk bandwidth and storage with reasonable amounts of buffer space, which is an impor-

*Work performed while visiting Bell Laboratories.

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the 23rd VLDB Conference**
**Athens, Greece, 1997**

tant cost factor in CMOD server design. The use of multiple disks to handle the high storage volume and bandwidth requirements of CM data exacerbates the problem. Thus, the need for intelligent scheduling mechanisms becomes more pronounced as the scale of the system increases.

A number of schemes for organizing CM data on multiple disks has been proposed in the literature [2, 3, 21, 23]. However, the applicability of these data layout schemes to EPPV service remains an open problem. The *matrix-based scheme* was designed to support periodic video retrieval for a given period while minimizing video buffering requirements [10]. Extensions to the base scheme that deal with the varying transfer rates of commonly used SCSI disks and different video display rates were presented in [12, 13]. However, the issue of videos with different retrieval periods was not addressed in any of these papers. Only in very recent work, Özden et al. [14] presented schemes for the periodic retrieval of videos from disk arrays using striping. Their work, however, addressed only a restricted form of the EPPV resource scheduling problems that assumes all clips to have *identical* display rates. Furthermore, they assume specific conditions on the video lengths that limit the usefulness of their results.

In this paper, we address the resource scheduling problems associated with supporting EPPV service in their most general form. We present a scheduling framework that handles continuous media data with (possibly) different display rates, different periods, and arbitrary lengths. Given a hardware configuration and a collection of clips to be scheduled, we present schemes for determining a schedulable subset of clips under different assumptions about data layout:

- **Clustering.** Each disk is viewed as an independent storage unit; that is, the data of each clip is stored on a single disk and multiple clips can be clustered on each disk.
- **Striping.** Each clip is declustered over all available disks.

In each case, our objective is to maximize the amount of disk bandwidth that is effectively scheduled. This is typically the situation facing large-scale CMOD servers that occasionally need to re-schedule their offerings to adapt to a changing audience, content, and popularity profile [8, 16]. For the clustering scheme, we formulate these optimization problems as generalized variants of the 0/1 knapsack problem [7, 18]. Since the problems are clearly $\mathcal{NP}$-hard, we present provably near-optimal heuristics with low polynomial-time complexity. We then present two alternative striping schemes. *Vertical Striping (VS)* views the entire disk array as a single large disk in a manner similar to fine-grained striping [11]. Despite its conceptual simplicity, VS suffers from increased disk latency overheads that render it impractical, especially for large disk arrays. *Horizontal Striping (HS)* is based on a round-robin distribution of clip data across the disks and has the potential of offering much better scalability and disk utilization than VS. This, however, comes at the cost of the more sophisticated scheduling methods required to support periodic stream retrieval. Specifically, we demonstrate that the scheduling problems involved in supporting EPPV service under HS are non-trivial generalizations of the Periodic Maintenance Scheduling Problem (PMSP) [22]. Given that PMSP is known to be $\mathcal{NP}$-complete in the strong sense [1], we propose novel heuristic algorithms for scheduling the periodic retrieval of horizontally striped clips. We follow a two-step approach. First, we introduce the novel concept of a *scheduling tree* structure and demonstrate its use in obtaining collision-free schedules for Periodic Maintenance. Next, we extend our methods to handle the more complex problems introduced by periodic retrieval under HS. Thus, our work also contributes to hard real-time scheduling theory by proposing the scheduling tree structure and algorithms as a new approach to Periodic Maintenance. Finally, we present preliminary experimental results that confirm the superiority of our HS-based scheme.

All theoretical results in this paper are stated without proof due to space constraints. The full proofs, as well as some interesting extensions to the ideas and results presented here, can be found in the full version of the paper [5].

# 2 Notation and System Model

Table 2 summarizes the notation used in this paper with a brief description of its semantics. Additional notation will be introduced when necessary. To avoid introducing data layout issues for multiple disks, we assume a single-disk server for the purposes of this section. The extension to multi-disk servers is straightforward once the data layout strategy (clustering, VS, HS) is specified.

Table 1: Clip and Disk Parameters

| Param. | Semantics |
|---|---|
| $C_i$ | Continuous media clip ($i = 1, \ldots, N$) (also, task of retrieving $C_i$) |
| $r_i$ | Display rate for clip $C_i$ (in Mbps) |
| $T_i$ | Retrieval period for clip $C_i$ (in sec) |
| $T$ | Time unit of clip retrieval (round length) |
| $l_i$ | Length of clip $C_i$ (in sec) |
| $n_i$ | Retrieval period of $C_i$ in rounds |
| $c_i$ | Number of columns in the matrix of $C_i$ |
| $d_i$ | Maximum column size (in bits) |
| $n_{disk}$ | Number of disks in CMOD server |
| $r_{disk}$ | Disk transfer rate |
| $c_{disk}$ | Disk storage capacity |
| $t_{seek}$ | Disk seek time |
| $t_{lat}$ | Disk latency |

## 2.1 Retrieving Continuous Media Data

We assume that the disk has a transfer rate of $r_{disk}$, a storage capacity of $c_{disk}$, a (worst case) seek time of $t_{seek}$, and a (worst case) latency of $t_{lat}$ (which consists of rotational delay and settle time). A clip $C_i$ is characterized by a display rate $r_i$ (the rate at which data for $C_i$ must be transmitted to clients) and a length $l_i$ (in units of time). We refer to the transmission of a clip starting at a given time as a *stream*. Data for streams is retrieved from the disk in *rounds of length* $T$. For a stream displaying clip $C_i$ (denoted by $stream(C_i)$), a circular buffer of size $2 \cdot T \cdot r_i$ is reserved in the server's buffer cache. In each round, while the stream is consuming $T \cdot r_i$ bits of data from its buffer, the $T \cdot r_i$ bits that the stream will consume in the next round are retrieved from the disk.

During a round, for streams $stream(C_1), \ldots, stream(C_k)$ for which data is to be retrieved from disk, $T \cdot r_1, ..., T \cdot r_k$ bits are read using the C-SCAN disk head scheduling algorithm [19]. C-SCAN ensures that the disk heads move in a single direction when servicing streams during a round. As a result, random seeks are eliminated and the total seek overhead during a round is bounded by $2 \cdot t_{seek}$. Furthermore, retrieval of each non-contiguously stored piece of data can incur a disk latency overhead of at most $t_{lat}$ during a round. To ensure that no stream starves during a round, the sum of the total disk transfer time for all data retrieved

and the overall latency and seek time overhead cannot exceed the length $T$ of the round [11, 17]. More formally, we require the following inequality to hold:

$$\sum_{\{stream(C_i)\}} (\frac{T \cdot r_i}{r_{disk}} + t_{lat}) + 2 \cdot t_{seek} \leq T. \quad (1)$$

### 2.2 Matrix-Based Allocation

EPPV service associates with each clip $C_i$ a retrieval period $T_i$ that is the reciprocal of its display frequency. We assume that retrieval periods are multiples of the round length $T$. This is a reasonable assumption, since retrieval periods will typically be multiples of minutes or even hours and the length of a round (usually bounded by buffering constraints) will not exceed a few seconds. *Matrix-based allocation* [10, 13], increases the number of clients that can be serviced under EPPV by laying out data based on the knowledge of retrieval periods[1]. The idea is to distribute, for each clip $C_i$, the starting points for the $\lceil \frac{l_i}{T_i} \rceil$ concurrent *display phases* of $C_i$ uniformly across its length. Each phase corresponds to a different stream servicing multiple clients. Conceptually, $C_i$ is viewed as a matrix consisting of elements of length $T$ (Figure 1(a)).

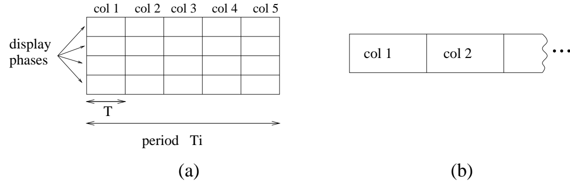



Figure 1: (a) A clip matrix. (b) Its layout on disk.

We define $n_i = \frac{T_i}{T}$ (i.e., the length of the retrieval period of $C_i$ in rounds). The matrix for $C_i$ consists of $c_i = \min\{n_i, \lceil \frac{l_i}{T} \rceil\}$ columns and $\lceil \frac{l_i}{T_i} \rceil$ rows (corresponding to the clip's display phases). Note that we can have $c_i < n_i$ when the retrieval period of the clip exceeds its length (i.e., $l_i < T_i$). Finally, we let $d_i$ denote the amount of data in a column of $C_i$'s matrix, that is[2] $d_i = \lceil \frac{l_i}{T_i} \rceil \cdot T \cdot r_i$.

To support periodic retrieval, a clip matrix is stored in column-major form and its retrieval is performed *in columns* (i.e., one column per round) with each element handed to a different display phase (Figure 1(b)). Matrix-based allocation reduces the overhead of disk latency per stream since, in each round, it incurs a total overhead of only $t_{lat}$ for $\lceil \frac{l_i}{T_i} \rceil$ streams of $C_i$, rather than $\lceil \frac{l_i}{T_i} \rceil \cdot t_{lat}$ (using Formula (1)). This means that the matrix-based scheme can support the periodic retrieval of $C_1, \ldots, C_k$ provided that the following inequality holds:

$$\sum_{\{C_i\}} (\frac{d_i}{r_{disk}} + t_{lat}) + 2 \cdot t_{seek} \leq T. \quad (2)$$

The disk bandwidth *effectively utilized* by a clip during a round is the amount of raw disk bandwidth consumed by the clip without accounting for the latency overhead. For $C_i$, this is exactly $\frac{d_i}{T}$, or, equivalently, $\lceil \frac{l_i}{T_i} \rceil \cdot r_i$.

[1] The scheduling algorithms presented in this paper can also be used with other data layout schemes. The interested reader is referred to the full version of the paper [5].

[2] Although some columns may actually contain less data than $d_i$ [13], in this paper, we are ignoring possible optimizations for smaller columns.

# 3 Clustering

Clustering views each disk as an autonomous unit – entire clips are stored on and retrieved from a single disk and multiple clips can be clustered on each disk. In this scenario, the EPPV resource scheduling problem reduces to effectively mapping clip matrices onto the server's disks so that the bandwidth and storage requirements of each matrix are satisfied. That is, the inequalities $\sum_i (\frac{d_i}{r_{disk}} + t_{lat}) + 2 \cdot t_{seek} \leq T$ and $\sum_i l_i \cdot r_i \leq c_{disk}$ need to hold for each disk, where the summation is taken over all clips $C_i$ stored on that disk. We address this scheduling problem in two stages. First, we present a solution that considers only the bandwidth requirements of clips. Next, we extend our approach to handle disk storage limitations. We present the first case separately since our results for this case will prove useful later in the paper, when striping is introduced.

### 3.1 Bandwidth Constraint

We associate two key parameters with each clip:

- A *size*: $\mathsf{size}(C_i) = \frac{\frac{d_i}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}$, that captures the normalized contribution of $C_i$ to the length of a round, or, equivalently, its (normalized) disk bandwidth consumption (see Formula (2)); and,
- A *value*: $\mathsf{value}(C_i) = \lceil \frac{l_i}{T_i} \rceil \cdot r_i$, that corresponds to the bandwidth *effectively utilized* by $C_i$ during a round.

Using these definitions, the problem of maximizing the effectively scheduled disk bandwidth can be formally stated as follows: *Given a collection of clips $C = \{C_1, \ldots, C_N\}$, determine a subset $C'$ of $C$ and a packing of* $\{\mathsf{size}(C_i) : C_i \in C'\}$ *in $n_{disk}$ unit capacity bins such that the total value* $\sum_{C_i \in C'} \mathsf{value}(C_i)$ *is maximized*. This problem is a generalization of the traditional 0/1 knapsack optimization problem (which can be seen as a special case with $n_{disk} = 1$) [7, 18]. Thus, it is clearly $\mathcal{NP}$-hard. Given the intractability of the problem, we present a fast heuristic algorithm (termed PACKCLIPS) that combines the value density heuristic rule for the classical knapsack problem [4] with a First-Fit packing rule. We define the value density of clip $C_i$ as the ratio $p_i = \frac{\mathsf{value}(C_i)}{\mathsf{size}(C_i)}$. Algorithm PACKCLIPS is depicted in Figure 2. The following lemma provides an upper bound on the worst-case performance ratio of our heuristic.

**Lemma 3.1** Algorithm PACKCLIPS runs in time $O(N(\log N + n_{disk}))$ and is 1/2-approximate; that is, if $V_{OPT}$ is the value of the optimal schedulable subset and $V_H$ is the value of the subset returned by PACKCLIPS then $\frac{V_H}{V_{OPT}} \geq \frac{1}{2}$. □

### 3.2 Bandwidth and Storage Constraints

We now extend the PACKCLIPS algorithm to handle the storage capacity constraints imposed by disks. The idea is to define the size of a clip $C_i$ as a *2-dimensional size vector* $\mathbf{s}_i = [\mathsf{size}_1(C_i), \mathsf{size}_2(C_i)]$, where the first component is the normalized bandwidth consumption of the clip (as defined in the previous section) and the second component is the normalized storage capacity requirement of the clip. More formally, $\mathsf{size}_1(C_i) = \frac{\frac{d_i}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}$ and $\mathsf{size}_2(C_i) = \frac{l_i \cdot r_i}{c_{disk}}$.

**Algorithm** PACKCLIPS($C, n_{disk}$)

**Input:** A collection of CM clips $C = \{C_1, \ldots, C_N\}$ and a number of disks $n_{disk}$.

**Output:** $C' \subseteq C$ and a packing of $C'$ in $n_{disk}$ unit capacity bins.(Goal: Maximize $\sum_{C_i \in C'} \mathsf{value}(C_i)$.)

1. Sort the clips in $C$ in non-increasing order of value density to obtain a list $L = < C_1, \ldots, C_N >$ where $p_i \geq p_{i+1}$. Initialize $\mathsf{load}(B_j) = \mathsf{value}(B_j) = 0$, $B_j = \emptyset$, for each bin (i.e., disk) $B_j$, $j = 1, \ldots, N$.
2. For each clip $C_i$ in L (in that order)
   - 2.1. Let $B_j$ be the first bin (i.e., disk) such that $\mathsf{load}(B_j) + \mathsf{size}(C_i) \leq 1$.
   - 2.2. Set $\mathsf{load}(B_j) = \mathsf{load}(B_j) + \mathsf{size}(C_i)$, $\mathsf{value}(B_j) = \mathsf{value}(B_j) + \mathsf{value}(C_i)$, $B_j = B_j \cup \{C_i\}$, and $L = L - \{C_i\}$.
3. Let $B_{<i>}$, $i = 1, \ldots, n_{disk}$ be the bins with the $n_{disk}$ largest $\mathsf{value}$'s in the final packing. Return $C' = \cup_{i=1}^{n_{disk}} B_{<i>}$. (The packing of $C'$ is defined by the $B_{<i>}$'s.)

Figure 2: Algorithm PACKCLIPS

Let $l(\mathbf{v})$ denote the maximum component of a vector $\mathbf{v}$ (i.e., its *length*). The 2-dimensional extension of the PACKCLIPS algorithm is based on defining the value density of a clip as the ratio $p_i = \frac{\mathsf{value}(C_i)}{l(\mathbf{s}_i)}$. The $\mathsf{load}$ of a disk is also a 2-dimensional vector equal to the vector sum of sizes of all clips clustered on that disk, and the condition in step 2.1 of PACKCLIPS becomes: $l(\mathsf{load}(B_j) + \mathbf{s}_i) \leq 1$. That is, we require that *both* the bandwidth and storage load on each disk do not exceed the disk's capacities. For our worst-case analysis of the 2-dimensional PACKCLIPS algorithm we also assume that the storage requirements of a clip never exceed one half of a disk's storage capacity, that is, $\mathsf{size}_2(C_i) \leq \frac{1}{2}$. This is a reasonable assumption since current disk storage capacities are in the order of several gigabytes. The following lemma shows that the extra dimension degrades the worst-case performance guarantee of our heuristic by a factor of two.

**Lemma 3.2** Assuming that the storage requirements of any clip are always less than or equal to one half of a disk's storage capacity, the 2-dimensional PACKCLIPS heuristic is 1/4-approximate; that is, if $V_{OPT}$ is the value of the optimal schedulable subset and $V_H$ is the value of the subset returned by PACKCLIPS then $\frac{V_H}{V_{OPT}} \geq \frac{1}{4}$. □

# 4 Disk Striping

A major deficiency of clustered data layout for large-scale EPPV service is that it can lead to severe disk storage and bandwidth fragmentation, and, consequently, underutilization of server resources. This problem is demonstrated in the rather discouraging worst-case bound of Lemma 3.2 – for "bad" lists of clips, PACKCLIPS may be able to utilize only as little as one fourth of the raw server capacity. Striping schemes eliminate storage fragmentation by declustering a clip's data across all available disks. In this section, we consider EPPV service under two distinct striping strategies termed *Vertical* and *Horizontal Striping*. Since storage fragmentation is no longer an issue, we can effectively ignore storage constraints by assuming that the aggregate storage requirements of the clips to be scheduled do not exceed the storage capacity of the server; that is, we assume that $\sum_i l_i \cdot r_i \leq n_{disk} \cdot c_{disk}$.

## 4.1 Vertical Striping (VS)

In the Vertical Striping scheme, each column of the clip matrix is declustered across all $n_{disk}$ disks of the server (Figure 3(a)). This scheme is similar to fine-grained striping [11] or RAID-3 data organization [15], since each column of the clip has to be retrieved in parallel from all disks (as a unit). Using the VS layout for clip matrices, implies that each disk is responsible for retrieving $\frac{1}{n_{disk}}$ of a clip's column in each round. Thus, the following condition must be satisfied on each disk:

$$\sum_{i=1}^{N} \frac{d_i}{r_{disk} \cdot n_{disk}} + N \cdot t_{lat} \leq T - 2 \cdot t_{seek}. \qquad (3)$$

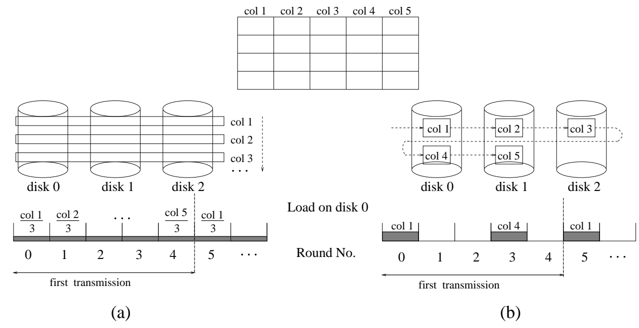



Figure 3: (a) Vertical Striping. (b) Horizontal Striping.

To ensure continuous retrieval under VS, all disks in the system must satisfy the same condition (namely, Formula (3)). Consequently, the problem of maximizing the effectively scheduled bandwidth clips under the VS scheme corresponds to a traditional, single-bin, 0/1 knapsack problem with clip sizes $\mathsf{size}(C_i) = \frac{\frac{d_i}{r_{disk} \cdot n_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}$ (from Ineq. (3)), and values $\mathsf{value}(C_i) = \lceil \frac{l_i}{T_i} \rceil \cdot r_i$ (as in Section 3). Thus, PACKCLIPS (with number of bins/disks equal to 1) readily provides a near-optimal heuristic for resource scheduling under VS.

Despite its conceptual and algorithmic simplicity, VS can lead to underutilization of available disk bandwidth due to increased latency overheads. This is because, during each round, all disks incur a penalty of $t_{lat}$ for each clip stored *in the entire server*. These latency penalties obviously limit the scalability of a VS-based EPPV server.

## 4.2 Horizontal Striping (HS)

In the Horizontal Striping scheme, the columns of a clip matrix are mapped to individual disks in a round-robin manner (Figure 3(b)). Consequently, the retrieval of data for a transmission of $C_i$ proceeds in a round-robin fashion along the disk array. During each round a single disk is used to read a column of $C_i$ and consecutive rounds employ consecutive disks[3].

Consider the retrieval of a clip matrix $C_i$ from a particular disk in the array. By virtue of the round-robin placement, during each

[3] We assume that a disk has sufficient bandwidth to support the retrieval of one or more clip columns. If this does not hold, one or more disks can be viewed as a single composite disk.

transmission of $C_i$, a column of $C_i$ must be retrieved from that disk periodically, at intervals of $n_{disk}$ rounds. From Formula (2), each such retrieval requires a fraction $\frac{\frac{d_i}{r_{disk}}+t_{lat}}{T-2\cdot t_{seek}}$ of the disk's bandwidth. Furthermore, to support EPPV service, the transmissions of $C_i$ are themselves periodic with a period $T_i = n_i \cdot T$.

Thus, the retrieval of a clip matrix $C_i$ from a specific disk in the array can be seen as a collection of *periodic real-time tasks* [9] with period $T_i$ (i.e., the clip's transmissions), where each task consists of a collection of *subtasks* that are $n_{disk} \cdot T$ time units apart (i.e., column retrievals within a transmission). Moreover, the computation time of each such subtask is $\frac{d_i}{r_{disk}} + t_{lat}$. An example of such a task is shown in Figure 3(b). Note that the maximum number of subtasks mapped to a disk by $C_i$ equals $\left\lceil \frac{c_i}{n_{disk}} \right\rceil$. ($c_i$ is the number of columns in $C_i$.) This number may actually be smaller for some disks in the array. However, in order to provide deterministic service guarantees for all disks, we consider only this worst-case number of subtasks in our scheduling formulation.

We say that two (or more) clip retrievals *collide* during a round if they are all reading data off the same disk. Collisions play a crucial role in our scheduling problem. Our algorithms need to ensure that whenever multiple retrievals collide during a round, their total bandwidth requirements do not exceed the capacity of the disk. For the simple case of two clips, we can use the *Generalized Chinese Remainder Theorem* [6] to prove the following lemma.

**Lemma 4.1** Consider two clips $C_1$ and $C_2$, and let $\alpha_i = \min\{\left\lceil \frac{c_i}{n_{disk}} \right\rceil, \frac{\gcd(n_1,n_2)}{\gcd(n_1,n_2,n_{disk})}\}$, $i = 1,2$. The retrieval of $C_1$ and $C_2$ can be scheduled without collisions *if and only if* $\alpha_1 + \alpha_2 \leq \gcd(n_1, n_2)$. □

Lemma 4.1 identifies a necessary and sufficient condition for the *collision-free* scheduling (or, *mergeability* [23]) of two clip retrieval patterns. Our result extends the result of Yu et al. [23] on merging two simple periodic patterns to the case of periodic tasks consisting of equidistant subtasks. Furthermore, Lemma 4.1 can be generalized to any number of clips if their periods can be expressed as $n_i = k \cdot m_i$ for all $i$, where $m_i$ and $m_j$ are relatively prime for all $i \neq j$. (For two clips, this condition is obviously true with $k = \gcd(n_1, n_2)$.)

**Lemma 4.2** Consider a collection of clips $C = \{C_1, \ldots, C_N\}$, with retrieval periods $n_i = k \cdot m_i$, for all $i$, where $\gcd(m_i, m_j) = 1$ for $i \neq j$. Let $\alpha_i = \min\{\left\lceil \frac{c_i}{n_{disk}} \right\rceil, \frac{k}{\gcd(k,n_{disk})}\}$. The retrieval of $C$ can be scheduled without collisions *if and only if* $\sum_{i=1}^{N} \alpha_i \leq k$. □

Unfortunately, Lemma 4.1 cannot be extended to the general case of multiple clips with arbitrary periods. In fact, in Section 5, we will show that deciding the existence of a collision-free schedule for the general case is $\mathcal{NP}$-complete in the strong sense. Thus, no efficient necessary and sufficient conditions are likely to exist. The condition described in Lemma 4.1 can easily be shown to be sufficient for no collisions in the general case. However, it is not necessary, as the following example indicates.

**Example 1:** Consider three clips with periods $n_1 = 4, n_2 = 6, n_3 = 8$ and let $n_{disk} = 4$. This set can be scheduled with no collisions, by initiating the retrieval of $C_1, C_2, C_3$ at rounds 0, 1, and 2, respectively. However, the inequality in Lemma 4.1 (extended for three clips) fails to hold, since $\gcd(n_1, n_2, n_3) = 2 < \sum_{i=1}^{3} \alpha_i = 3$.

# 5 The Scheduling Tree Structure

In this section, we address the problem of scheduling EPPV service under HS. We first consider a model of simple periodic real-time tasks and show that deciding the existence of a collision-free schedule is equivalent to *Periodic Maintenance* [1, 22], a problem known to be intractable. Motivated from this result, we define the novel concept of a *scheduling tree* and discuss its application in a heuristic algorithm for Periodic Maintenance. We then show how the scheduling tree structure can handle the more complex model of periodic tasks identified in Section 4.2.

## 5.1 Periodic Maintenance Scheduling

The *k-server Periodic Maintenance Scheduling Problem (k-PMSP)* [1] is a special case of the problem of scheduling simple periodic tasks in a hard real-time environment. Briefly, the $k$-PMSP decision problem can be stated as follows: *Let $C = \{C_1, \ldots, C_N\}$ be a set of periodic tasks with corresponding periods $P = \{n_1, \ldots, n_N\}$, where each $n_i$ is a positive integer. Is there a mapping of the the tasks in $C$ to positive integer time slots such that successive occurrences of $C_i$ are* **exactly** *$n_i$ time slots apart and no more than $k$ tasks ever collide in a slot?* Note that if $u_i$ is the index of the first occurrence of $C_i$ in a schedule for $P$ then the (multi)set of starting time slots $\{u_1, \ldots, u_N\}$ uniquely determines the schedule, since $C_i$ occurs at all slots $u_i + j \cdot n_i$, $j \geq 0$.

Baruah et al. [1] have shown that for any fixed value $k \geq 1$, $k$-PMSP is $\mathcal{NP}$-complete in the strong sense. Consequently, given a collection of simple periodic tasks with periods $P$, determining the existence of a collision-free schedule is intractable (i.e., it is equivalent to 1-PMSP). The existence of a *scheduling tree* structure (as described below) that contains all the periods in $P$, guarantees the existence of a collision-free schedule. Furthermore, the starting time slot for each task can be determined from the scheduling tree[4].

**Definition 5.1** A *scheduling tree* is a tree structure consisting of nodes and edges with integer weights, where:

1. Each internal node of weight $w$ can have *at most* $w$ outgoing edges, each of which has a distinct weight in $\{0, 1, \ldots, w-1\}$; and,
2. Each leaf node represents a period $n_i$ such that $n_i$ is equal to the product of weights of the leaf's ancestor nodes.

We define the *level of a node* (or, *edge*) as the number of its proper ancestor nodes. Thus the level of the tree's root is 0 and the level of all edges emanating from the root is 1. For any node $n$, let $w(n)$ and $e(n)$ denote the weight and the number of edges of $n$, respectively. Also, let $\mathsf{ancestor_node}_j(n)$ represent the weight of the ancestor node of $n$ at level $j$, and let $\mathsf{ancestor_edge}_j(n)$ denote the weight of the ancestor edge of $n$ at level $j$, where $j \leq level(n)$. Finally, define $\pi_j(n) = \prod_{i=0}^{j} \mathsf{ancestor_node}_i(n)$ for $0 \leq j \leq level(n)$.

Consider a leaf node for period $n_i$ located at level $l$. The first slot $u_i$ in which the corresponding task is scheduled is defined

[4] To the best of our knowledge, no similar notion of tree structure for periodic task scheduling has been proposed in the real-time scheduling literature [20].

from the scheduling tree structure as follows:

$$u_i = \mathsf{ancestor_edge}_1(n_i) + \sum_{j=2}^{l} \mathsf{ancestor_edge}_j(n_i) \cdot \pi_{j-2}(n_i). \quad (4)$$

Some intuition for the scheduling tree structure and the above formula is provided in Figure 4. The basic idea is that all tasks in a subtree rooted at some edge emanating from node $n$ at level $l$ will utilize time slot numbers that are congruent to $i \pmod{\pi_l(n)}$, where $i$ is a unique number between 0 and $\pi_l(n) - 1$. Satisfying this invariant recursively at every internal node ensures the avoidance of collisions.

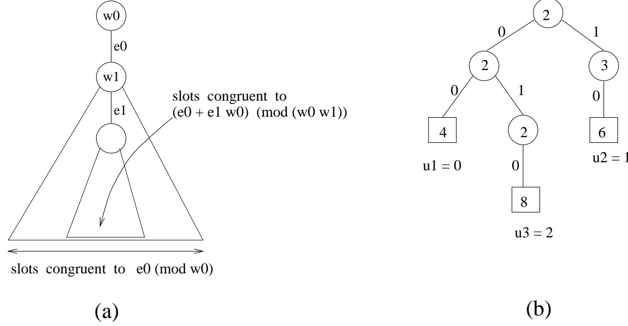



Figure 4: (a) The scheduling tree structure. (b) An example tree.

Note that the existence of a scheduling tree for a set of periods $P$ is only a *sufficient condition* for the existence of a collision-free schedule. For example, the periods 6, 10, and 15 are schedulable using start times of 0, 1, and 2, respectively, although no scheduling tree can be built (since $\gcd(\{6, 10, 15\}) = 1$). However, using the Generalized Chinese Remainder Theorem it is straightforward to show that the existence of a *scheduling forest*, as defined below, is both necessary and sufficient for the existence a collision-free schedule.

**Definition 5.2** Let $\Gamma_i$ denote a scheduling tree for $P_i$. The trees $\Gamma_i$ and $\Gamma_j$ are *consistent* if and only if for each $n_m \in P_i$ and $n_l \in P_j$ we have $u_m \not\equiv u_l \pmod{\gcd(n_m, n_l)}$. A *scheduling forest* for $P$ is a collection of pairwise consistent scheduling trees for some partitioning $P_1, \ldots, P_k$ of $P$.

**Lemma 5.1** Determining whether there exists a scheduling forest for $P$ is equivalent to 1-PMSP, and, thus, it is $\mathcal{NP}$-complete in the strong sense. □

Given the above intractability result, we present a heuristic algorithm for constructing scheduling trees for a given (multi)set of periods. Our algorithm is based on identifying and incrementally maintaining *candidate nodes* for scheduling incoming periods.

**Definition 5.3** An *internal* node $n$ at level $l$ is *candidate for period* $n_i$ if and only if $\pi_{l-1}(n) | n_i$ and $\gcd(w(n), \frac{n_i}{\pi_{l-1}(n)}) \geq \frac{w(n)}{w(n)-e(n)}$.

A period $n_i$ can be scheduled under any candidate node $n$ in a scheduling tree. There are two possible cases:

- **If** $\pi_l(n) | n_i$ then Definition 5.3 guarantees that $n$ has at least one free edge at which $n_i$ can be placed (Figure 5(a)).

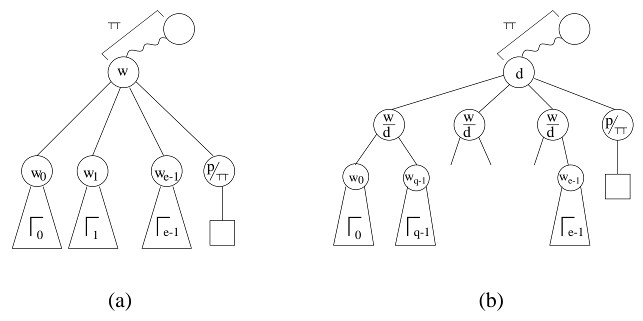



Figure 5: (a) Placing a period $p$ under a scheduling tree node without splitting. (b) Period placement when the node is split.

- **If** $\pi_l(n) \not| n_i$ then, in order to accommodate $n_i$ under node $n$, $n$ must be *split* so that the defining properties of the scheduling tree structure are kept intact. This is done as follows. Let $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$. Node $n$ is split into a parent node with weight $d$ and child nodes with weight $\frac{w(n)}{d}$, with the original children of $n$ divided among the new child nodes, as shown in Figure 5(b); that is, the first batch of $\frac{w(n)}{d}$ children of $n$ are placed under the first child node, and so on. It is easy to see that this splitting maintains the properties of the structure. Furthermore, Definition 5.3 guarantees that the new parent node has at least one free edge for scheduling $n_i$.

The set of candidate nodes for each period to be scheduled can be maintained efficiently, in an incremental manner. The observation here is that when a new period $n_i$ is scheduled, all remaining periods only have to check a maximum of three nodes, namely the two closest ancestors of the leaf for $n_i$ and, if a split occurred, the last child node created in the split, for possible inclusion or exclusion from their candidate sets.

As in Section 3, we assume each task is associated with a value and we aim to maximize the cumulative value of a schedule. The basic idea of our heuristic (termed BUILDTREE) is to build the scheduling tree incrementally in a greedy fashion, scanning the tasks in non-increasing order of value and placing each period $n_i$ in that candidate node $M$ that implies the minimum value loss among all possible candidates. This loss is calculated as the total value of all periods whose candidate sets become empty after the placement of $n_i$ under $M$. Ties are always broken in favor of those candidate nodes that are located at higher levels (i.e., closer to the leaves), while ties at the same level are broken using the postorder node numbers (i.e., left-to-right order). When a period is scheduled in $\Gamma$, the candidate node sets for all remaining periods are updated (in an incremental fashion) and the algorithm continues with the next task/period (with at least one candidate in $\Gamma$). Algorithm BUILDTREE is depicted in Figure 6.

Let $N$ be the number of tasks in $C$. The number of internal nodes in a scheduling tree is always going to be $O(N)$. To see this, note that an internal node will always have at least two children, with the only possible exception being the rightmost one or two new nodes created during the insertion of a new period (depending on whether splitting was used, see Figure 5). Since the number of insertions is at most $N$, it follows that the number of internal nodes is $O(N)$. Based on this fact, it is easy to show that BUILDTREE runs in time $O(N^3)$.

**Example 2:** Consider the list of periods $< n_1 = 2, n_2 = 12, n_3 = 30 >$ (sorted in non-increasing order of value). Figure 7

**Algorithm** BUILDTREE(**C,** value)

**Input:** A set of simple periodic tasks $C = \{C_1, \ldots, C_N\}$ with corresponding periods $P = \{n_1, \ldots, n_N\}$, and a value() function assigning a value to each $C_i$.

**Output:** A scheduling tree $\Gamma$ for a subset $C'$ of $C$. (Goal: Maximize $\sum_{C_i \in C'} \mathsf{value}(C_i)$.)

1. Sort the tasks in $C$ in non-increasing order of value to obtain a list $L =< C_1, C_2, \ldots, C_N >$, where $\mathsf{value}(C_i) \geq \mathsf{value}(C_{i+1})$. Initially, $\Gamma$ consists of a root node with a weight equal to $n_1$.
2. For each periodic task $C_i$ in $L$ (in that order)
   - 2.1. Let $cand(n_i, \Gamma)$ be the set of candidate nodes for $n_i$ in $\Gamma$. (Note that this set is maintained incrementally as the tree is built.)
   - 2.2. For each $n \in cand(n_i, \Gamma)$, let $\Gamma \cup \{n_i\}_n$ denote the tree that results when $n_i$ is placed under node $n$ in $\Gamma$. Let $\mathsf{loss}(n) = \{C_j \in L - \{C_i\} \mid cand(\Gamma \cup \{n_i\}_n) = \emptyset\}$ and $\mathsf{value}(\mathsf{loss}(n)) = \sum_{C_j \in \mathsf{loss}(n)} \mathsf{value}(C_j)$.
   - 2.3. Place $n_i$ under the candidate $M$ such that $\mathsf{value}(\mathsf{loss}(M)) = \min_{cand(n_i,\Gamma)}\{\mathsf{value}(\mathsf{loss}(n))\}$. (Ties are broken in favor of nodes at higher levels.) If necessary, node $M$ is split.
   - 2.4. Set $\Gamma = \Gamma \cup \{n_i\}_M$, $L = L - \mathsf{loss}(M)$.
   - 2.5. For each task $C_j \in L$, update the candidate node set $cand(n_j, \Gamma)$.

Figure 6: Algorithm BUILDTREE

illustrates the step-by-step construction of the scheduling tree using BUILDTREE. Note that period $n_3$ splits the node with weight 6 into two nodes with weights 3 and 2.

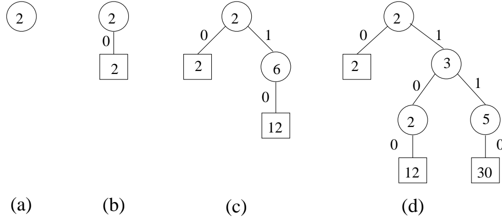



Figure 7: Construction of a scheduling tree.

## 5.2 Scheduling Equidistant Subtasks

In Section 4.2, we identified a clip retrieval under Horizontal Striping as a periodic real-time task $C_i$ with period $n_i = \frac{T_i}{T}$ (in rounds) that consists of a collection of $\left\lceil \frac{c_i}{n_{disk}} \right\rceil$ subtasks that need to be scheduled $n_{disk}$ rounds apart. The basic observation here is that all the subtasks of $C_i$ are themselves periodic with period $n_i$, so the techniques of the previous section can be used for each individual subtask. However, the scheduling algorithm also needs to ensure that *all* the subtasks are scheduled together, using time slots (i.e., rounds) placed regularly at intervals of $n_{disk}$. In this section, we propose heuristic methods for building a scheduling tree in this generalized setting.

An important requirement of this more general task model is that the insertion of new periods cannot be allowed to distort the relative placement of subtasks already in the tree. The splitting mechanism described in the previous section for simple periodic tasks does not satisfy this requirement, since it can alter the starting time slots for all subtasks located under the split node. We describe a new rule for splitting nodes without modifying the retrieval schedule for subtasks already in the tree. The idea is to use a different method for "batching" the children of the node being split, so that the starting time slots for all leaf nodes (as specified by Equation (4)) remain unchanged. This new splitting rule is as follows: *If the node $n$ is split to give a new parent node with weight $d$, then place at edge $i$ of the new node ($i = 0, \ldots, d-1$) all the children of the old node $n$ whose parent edge weight was congruent to $i \pmod d$.* Our claim that retrieval schedules are kept intact under this rule is a direct consequence of Equation (4).

**Example 3:** Figure 8(a) illustrates a scheduling tree with two tasks with periods $n_1 = 6$, $n_2 = 6$ assigned to slots 0 and 1. Figure 8(b) depicts the scheduling tree after a third task with period $n_3 = 15$ is inserted. Although there is enough capacity for both $n_1$ and $n_2$ in the subtree connected to the root with edge 0, the new split forces $n_2$ to be placed in the subtree connected to the root with edge 1.

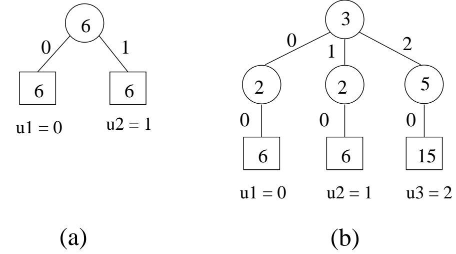



Figure 8: Illustration of the new splitting rule

In this setting, candidate nodes are defined as follows.

**Definition 5.4** An internal node $n$ at level $l$ is *candidate for period* $n_i$ if and only if $\pi_{l-1}(n) | n_i$ and there exists an $i \in \{0, \ldots, d-1\}$ such that all edges of $n$ with weights congruent to $i \pmod d$ are free, where $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$.

However, under our generalized model of periodic tasks, a candidate node for $n_i$ can only accommodate a subtask of $C_i$. This is clearly not sufficient for the entire task. The temporal dependency among the subtasks of $C_i$ means that our scheduling tree scheme must make sure that *all* the subtasks of $C_i$ are placed in the tree at distances of $n_{disk}$.

One way to deal with this situation is to maintain candidate nodes for subtasks based on Definition 5.4, and use a simple predicate based on Equation (4), for checking the availability of specific time slots in the scheduling tree. The scheduling of $C_i$ can then be handled as follows. Select a candidate node for $n_i$ and a time slot $u_i$ for $n_i$ under this candidate. Place the first subtask of $C_i$ in $u_i$ and call the predicate repeatedly to check if $n_i$ can be scheduled in slot $u_i + j \cdot n_{disk}$, for $j = 1, \ldots, \left\lceil \frac{c_i}{n_{disk}} \right\rceil$. If the predicate succeeds for all $j$, then $C_i$ is scheduled starting at $u_i$. Otherwise, the algorithm can try another potential starting slot $u_i$. In the full version of the paper [5], we describe a predicate for checking slot availability that can be used in this scheme.

A problem with the approach outline above is that even if the number of starting slots tried for $C_i$ is restricted to a constant, scheduling each subtask individually yields pseudo-polynomial

time complexity. This is because the number of scheduling operations in a trial will be $O(\frac{c_i}{n_{disk}})$, where $c_i = \min\{n_i, \frac{l_i}{T}\}$ is part of the problem input.

We propose a polynomial time heuristic algorithm for the problem. To simplify the presentation, we assume that every period $n_i$ is a multiple of $n_{disk}$. Although it is possible to extend our heuristic to handle general periods, we believe that this assumption is not very restrictive in practice. This is because we typically expect round lengths $T$ to be in the area of a few seconds and periods $T_i$ to be multiples of some number of minutes (e.g., 5, 10, 30, or 60 minutes). Therefore, it is realistic to assume the smallest period in the system can be selected to be a multiple of $n_{disk}$. Our goal is to devise a method that ensures that if the *first subtask* of a task $C_i$ does not collide with the first subtask of any other task in the tree, then no other combination of subtasks can cause a collision to occur. This means that once the first subtask of $C_i$ is placed in the scheduling tree there is no need to check the rest of $C_i$'s subtasks individually.

Our algorithm sets the weight of the root of the scheduling tree to $n_{disk}$. (This is possible since the $n_i$'s are multiples of $n_{disk}$.) By Equation (4), this implies that consecutive subtasks of a task will require consecutive edges emanating from nodes at the first level (i.e., the direct descendants of the root). The basic idea of our method is to make sure that when the first subtask of a task is placed at a leaf node, a number of consecutive edges of the first-level ancestor node of that leaf are *disabled*, so that the slots under those edges cannot be used by the first subtask of any future task. By our previous observation, $s_i - 1 = \lceil \frac{c_i}{n_{disk}} \rceil - 1$ consecutive edges of the first-level ancestor of the leaf for $n_i$ must be disabled, starting with the right neighbor of the edge under which that leaf resides. ($s_i$ is the number of subtasks of $C_i$.) This "edge disabling" is implemented by maintaining an integer *distance* for each edge $e$ emanating from a first-level node that is equal to the number of consecutive neighbors of $e$ that have been disabled. Our placement algorithm has to maintain two invariants. First, the distance of an edge $e$ of a first-level node is always equal to $\max_{C_i}\{s_i\} - 1$, where the maximum is taken over all tasks placed under $e$ in the tree. Second, the sum of the weight of an edge $e$ of a first-level node $n$ and its distance is always less than the weight of $n$ (so that the defining properties of the tree are maintained). The formal definition of our algorithm is omitted due to space constraints. The full details can be found in [5].

### 5.3 Handling Slots with Multi-Task Capacities

The scheduling tree formulation can easily be extended to handle time slots that can fit more than one subtask (i.e., can allow for some tasks to collide). As we saw in Section 4.2, this is exactly the case for the rounds of EPPV retrieval under HS. Using the notation of Section 3, we can think of the subtasks of $C_i$ as items of size $\mathsf{size}(C_i) \leq 1$ (i.e., the fraction of disk bandwidth required for retrieving one column of clip $C_i$) that are placed in unit capacity time slots. In this more general setting, a time slot can accommodate multiple tasks as long as their total size does not exceed one. Note that this problem is a generalization of the $k$-server Periodic Maintenance Scheduling Problem ($k$-PMSP), where all items are assumed to be of the same size (i.e., $\frac{1}{k}$th of the capacity).

The problem can be visualized as a collection of unit capacity bins (i.e., time slots) located at the leaves of a scheduling tree, whose structure determines the eligible bins for each task's subtasks (based on their period). With respect to our previous model of tasks, the main difference is that since slots can now accommodate multiple retrievals it is possible for a leaf node that is already occupied to be a candidate for a period. Hence, the basic idea for extending our schemes to this case is to keep track of the available slot space at each leaf node and allow leaf nodes to be shared by tasks. Thus, our notion of candidate nodes can simply be extended as follows.

**Definition 5.5** Let $n$ be a leaf node for of a scheduling tree $\Gamma$ corresponding to period $p$. Also, let $S(n)$ denote the collection of tasks (with period $p$) mapped to $n$. The *load of leaf* $n$ is defined as: $\mathsf{load}(n) = \sum_{C_i \in S(n)} \mathsf{size}(C_i)$.

**Definition 5.6** A node $n$ at level $l$ is *candidate for a task of* $C_i$ (with period $n_i$) if and only if:

1. $n$ is internal, conditions in Definition 5.4 hold, or
2. $n$ is external (leaf node) corresponding to $n_i$ (i.e., $\pi_l(n) = n_i$), and $\mathsf{load}(n) + \mathsf{size}(C_i) \leq 1$.

With these extensions, it is easy to see that the methods of Section 5.2 can be used without modification to produce a scheduling tree for the multi-task capacity case.

## 6 Combining Multiple Scheduling Trees

To construct forests of multiple non-colliding scheduling trees, trees already built can be used to restrict task placement in the tree under construction. By the Generalized Chinese Remainder Theorem, the scheduling algorithm needs to ensure that each subtask of task $C_i$ is assigned a slot $u_i$ such that $u_i \not\equiv u_j \pmod{\gcd(n_i, n_j)}$ for any subtask of any task $C_j$ that is scheduled in slot $u_j$ in a previous tree within the same forest. This obviously is a very expensive method and efficient heuristics for constructing scheduling forests still elude our efforts. In this section, however, we provide a general packing-based scheme that can be used for combining independently built scheduling forests. Of course, for our purposes, a forest can always consist of a single tree. Our goal is to improve the utilization of scheduling slots that can accommodate multiple tasks.

Given a collection of tasks, scheduling forests are constructed until each task is assigned a time slot. We know that no pair of tasks within a forest will collide at any slot except for tasks with the same period that are assigned to the same leaf node as described in Section 5.3. A simple conservative approach is to assume a worst-case collision across forests. That is, we define the size of a forest as $\mathsf{size}(F_i) = \max_{n_j \in F_i}\{\mathsf{load}(n_j)\}$ where $n_j$ is any leaf node in $F_i$, and the load of a leaf node is as in Definition 5.5. Further, a forest $F_i$ has a value: $\mathsf{value}(F_i) = \sum_{C_j \in F_i} \mathsf{value}(C_j)$. Thus, under the assumption of a worst-case collision, the problem of maximizing the total scheduled value for a collection of forests is a traditional 0/1 knapsack optimization problem. A packing-based heuristic like PACKCLIPS can be used to provide an approximate solution.

In some cases, the worst-case collision assumption across forests may be unnecessarily restrictive. For example, consider two scheduling trees $\Gamma_1$ and $\Gamma_2$ that are constructed independently. Let $e_1$ be an edge emanating from the root node $n_1$ of $\Gamma_1$ and $e_2$ be an edge emanating from the root node $n_2$ of $\Gamma_2$. If $e_1 \bmod (\gcd(n_1, n_2)) \neq e_2 \bmod (\gcd(n_1, n_2))$ holds, then the tasks scheduled in the subtrees rooted at $e_1$ and $e_2$ can never collide. Using such observations, we can devise more clever packing-based schemes for combining forests [5].

# 7 Experimental Performance Evaluation

## 7.1 Experimental Testbed

For our experiments, we used two basic workload components, modeling typical scenarios encountered in today's pay-per-view video servers.

- **Workload #1** consisted of relatively long MPEG-1 compressed videos with a duration between 90 and 120 minutes (e.g., movie features). The display rate for all these videos was equal to $r_i$ = 1.5 Mbps. To model differences in video popularity, our workload comprised two distinct regions: a "hot region" with retrieval periods selected randomly between 40 and 60 minutes and a "cold region" with periods between 150 and 180 minutes. Different type #1 workloads were generated by varying the size of the hot region between 5% and 50% of the total number of clips.
- **Workload #2** consisted of small video clips with lengths between 2 and 10 minutes (e.g., commercials or music video clips). The display rates for these videos varied between 2 and 4 Mbps (i.e., MPEG-1 and 2 compression). Again, clips were divided between a "hot region" with periods selected randomly between 20 and 30 minutes and a "cold region" with periods between 40 and 60 minutes. Different type #2 workloads were generated by varying the size of the hot region between 5% and 50% of the total number of clips.

We experimented with each component executing in isolation and with mixed workloads consisting of mixtures of type #1 and type #2 workloads. We concentrated on scaleup experiments in which the total expected storage requirements of the offered workload were approximately equal to the total storage capacity of the server. This allowed us to effectively ignore the storage capacity constraint for the striping-based schemes. For clustering, storage capacities were accounted for by using the 2-dimensional version of PACKCLIPS (Section 3.2). Our basic performance metric was the *effectively scheduled disk bandwidth* (in Mbps) for each of the resource scheduling schemes presented in this paper. (The graphs presented in the next section are indicative of the results obtained over the ranges of the workload parameters.)

The results discussed in this paper were obtained assuming a bandwidth capacity of $r_{disk}$ = 80 Mbps and a storage capacity of $c_{disk}$ = 4 GBytes for each disk in the server. The (worst-case) disk seek time and latency were set at $t_{seek}$ = 24 ms and $t_{lat}$ = 9.3 ms, respectively, and the round length was $T$ = 1 sec. As part of our future work, we plan to examine the effect of these parameters on the performance of our scheduling schemes.

## 7.2 Experimental Results

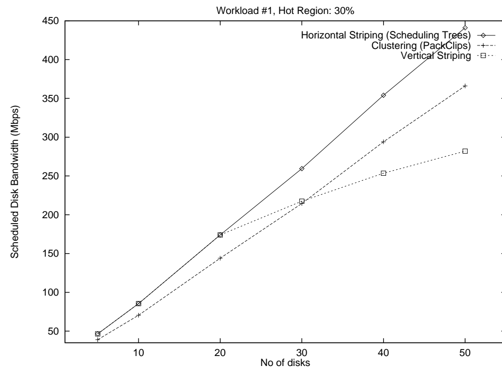
The results of our experiments with type #1 workloads with hot regions of 30% and 10% are shown in Figures 9(a) and 9(b), respectively. Clearly, the HS-based scheme outperforms both clustering and VS over the entire range of values for the number of disks. Observe that for type #1 workloads and for the disk parameter values used in our study, the maximum number of clips that can be scheduled is limited by the aggregate disk storage. Specifically, it is easy to see that the maximum number of clips that can fit in a disk is 3.95 and the average number of concurrent streams for a clip is $(0.3 \cdot 3 + 0.7 \cdot 1) = 1.6$. Thus the maximum bandwidth that can be utilized on a single disk for this mix of accesses is $1.6 \cdot 3.95 \cdot 1.5 = 9.48$Mbps. This explains the low scheduled bandwidth output shown in Figure 9. We should note that in most cases our scheduling tree heuristics were able to schedule the entire offered workload of clips. On the other hand, the performance of VS schemes quickly deteriorates as the size of the disk array increases. This confirms our remarks on the limited scalability of VS in Section 4.1. The performance of our clustering scheme under Workload #1 suffers from the disk storage fragmentation due to the large clip sizes. We also observe a deterioration in the performance of clustering as the access skew increases (i.e., the size of the hot region becomes smaller). This can be explained as follows: PACKCLIPS first tries to pack the clips that give the highest profit (i.e., the hot clips). Thus when the hot region becomes smaller the relative value of the scheduled subset (as compared to the total workload value) decreases.

The relative performance of the three schemes for a type #2 workload with a 50% hot region is depicted in Figure 10(a). Again, the HS-based scheme outperforms both clustering and VS over the entire range of $n_{disk}$. Note that, compared to type #1 workloads, the relative performance of clustering and VS schemes under this workload of short clips is significantly worse. This is because both these schemes, being unaware of the periodic nature of clip retrieval, reserve a specific amount of bandwidth for every clip $C_i$ during every round of length $T$. However, for clips whose length is relatively small compared to their period this bandwidth will actually be needed only for small fraction of rounds. Figure 10(a) clearly demonstrates the devastating effects of this bandwidth wastage and the need for periodic scheduling algorithms.

Finally, Figure 10(b) depicts the results obtained for a mixed workload consisting of 30% type #1 clips and 70% type #2 clips. HS is once again consistently better than VS and clustering over the entire range of disk array sizes. Compared to pure type #1 or #2 workloads, the clustering-based scheme is able to exploit the non-uniformities in the mixed workload to produce much better packings. This gives clustering a clear win over VS. Still, its wastefulness of disk bandwidth for short clips does not allow it to perform at the level of HS.

# 8 Conclusions

In this paper we have addressed the resource scheduling and data organization problems associated with supporting EPPV service in their most general form; that is, for clips with possibly different display rates, periods, lengths. We studied three different approaches to utilizing multiple disks: clustering, vertical striping (VS) and horizontal striping (HS). In each case, the periodic nature of the EPPV service model raises a host of interesting resource scheduling problems. For clustering and VS, we presented a knapsack formulation that allowed us to obtain a provably near-optimal heuristic with low polynomial time complexity. However, both these data layout schemes have serious drawbacks: Clustering can suffer from severe storage and bandwidth fragmentation, and VS incurs high disk latency overheads that limit its scalability. HS, on the other hand, avoids these problems but requires sophisticated hard real-time scheduling methods to support periodic retrieval. Specifically, we showed the EPPV scheduling problem for HS to be a generalization of the Periodic Maintenance Scheduling Problem [22] and developed a number of novel concepts and algorithmic solutions to address the issues involved. Finally, we presented a preliminary set of experimental results that verified our expectations about the average performance of the three schemes: Clus-



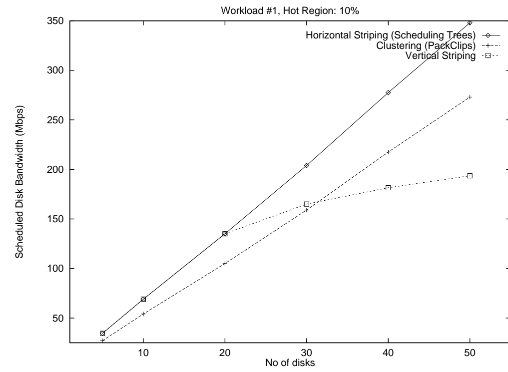
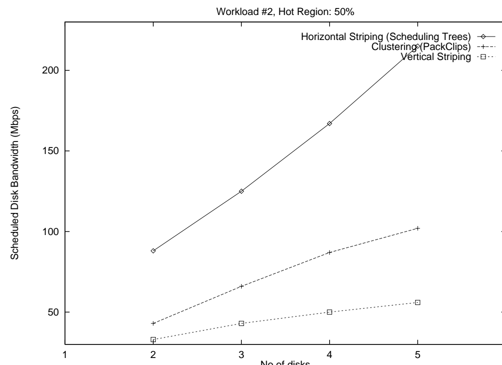




Figure 9: (a) Workload #1, 30% hot. (b) Workload #1, 10% hot.



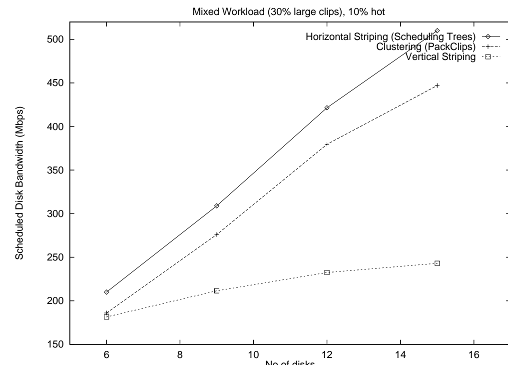




Figure 10: (a) Workload #2, 50% hot. (b) Mixed Workload (30%-70%), 10% hot.

tering can lead to fragmentation and underutilization of resources and the performance of VS does not scale linearly in the number of disks due to increased latencies. Our novel tree-based algorithm for HS emerged as the clear winner under a variety of randomly generated workloads.

## References


[1] S. Baruah, L. Rosier, I. Tulchinsky, and D. Varvel. "The Complexity of Periodic Maintenance". In *Proc. of the 1990 Intl. Computer Symp.*, Taiwan, 1990.

[2] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. "Staggered Striping in Multimedia Information Systems". In *Proc. of the 1994 ACM SIGMOD Intl. Conf.*, May 1994.

[3] M.-S. Chen, D. D. Kandlur, and P. S. Yu. "Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams". In *Proc. of ACM Multimedia '93*, August 1993.

[4] M.R. Garey and D.S. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. W.H. Freeman, 1979.

[5] M. N. Garofalakis, B. Özden, and A. Silberschatz. "Resource Scheduling in Enhanced Pay-Per-View Continuous Media Databases". Tech. Memorandum BL0112330-970107-01, Bell Laboratories, 1997.

[6] D. E. Knuth. *"The Art of Computer Programming (Vol. 2 / Seminumerical Algorithms)"*. Addison-Wesley, 1981.

[7] E. L. Lawler. "Fast Approximation Algorithms for Knapsack Problems". *Math. of Operations Research*, 4(4):339–356, 1979.

[8] T.D.C. Little and D. Venkatesh. "Popularity-Based Assignment of Movies to Storage Devices in a Video-on-Demand System". *ACM Multimedia Systems*, 2:280–287, 1995.

[9] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". *Journal of the ACM*, 20(1):46–61, 1973.

[10] B. Özden, A. Biliris, R. Rastogi, and A. Silberschatz. "A Low-Cost Storage Server for Movie on Demand Databases". In *Proc. of the 20th Intl. VLDB Conf.*, September 1994.

[11] B. Özden, R. Rastogi, and A. Silberschatz. "Disk Striping in Video Server Environments". *IEEE Data Engineering Bulletin*, 18(4):4–16, 1995.

[12] B. Özden, R. Rastogi, and A. Silberschatz. "On the Design of a Low-Cost Video-on-Demand Storage System". *ACM Multimedia Systems*, 4:40–54, 1996.

[13] B. Özden, R. Rastogi, and A. Silberschatz. "The Storage and Retrieval of Continuous Media Data". In *"Multimedia Database Systems: Issues and Research Directions", V.S. Subrahmanian and S. Jajodia (Eds.)*. Springer-Verlag, 1996.

[14] B. Özden, R. Rastogi, and A. Silberschatz. "Periodic Retrieval of Videos from Disk Arrays". In *Proc. of the 13th Intl. Conf. on Data Engineering*, April 1997.

[15] D. A. Patterson, G. A. Gibson, and R. H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". In *Proc. of the 1988 ACM SIGMOD Intl. Conf.*, June 1988.

[16] PRECEPT Software, Inc. IP/TV Datasheets. (`http://www.precept.com/datasheets/html/iptvds1.htm`).

[17] P. V. Rangan and H. M. Vin. "Efficient Storage Techniques for Digital Continuous Multimedia". *IEEE Trans. on Knowledge and Data Engineering*, 5(4):564–573, 1993.

[18] S. Sahni. "Approximate Algorithms for the 0/1 Knapsack Problem". *Journal of the ACM*, 22(1):115–124, 1975.

[19] A. Silberschatz and P. Galvin. *"Operating System Concepts"*. Addison-Wesley, 1994.

[20] J. A. Stankovic and K. Ramamritham, eds. *"Advances in Real-Time Systems"*. IEEE Computer Society Press, 1993.

[21] F. A. Tobagi, J. Pang, R. Baird, and M. Gang. "Streaming RAID: A Disk Storage System for Video and Audio Files". In *Proc. of ACM Multimedia '93*, August 1993.

[22] W.D. Wei and C.L. Liu. "On a Periodic Maintenance Problem". *Operations Research Letters*, 2(2):90–93, 1983.

[23] C. Yu, W. Sun, D. Bitton, Q. Yang, R. Bruno, and J. Tullis. "Efficient Placement of Audio Data on Optical Disks for Real-Time Applications". *Comm. of the ACM*, 32(7):862–871, 1989.