The VLDB Journal (1998) 7: 206–225

The VLDB Journal
© Springer-Verlag 1998

On periodic resource scheduling for continuous-media databases

Minos N. Garofalakis, Banu Özden, Avi Silberschatz

Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA; E-mail: {minos,ozden,avi}@research.bell-labs.com

Edited by. T. Özsu and S. Christodoulakis. Received June 9, 1998 / Accepted October 13, 1998

Abstract. The Enhanced Pay-Per-View (EPPV) model for providing continuous-media 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. The EPPV model offers a number of advantages over other data-sharing schemes (e.g., batching), which make it more attractive to large-scale service providers. 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. Our 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. We also present preliminary experimental results for the average case behavior of the proposed scheduling schemes and examine how they compare to each other under different workloads. A major contribution of our work is the introduction of a robust scheduling framework that, we believe, can provide solutions for a variety of realistic EPPV resource-scheduling scenarios, as well as any scheduling problem involving regular, periodic use of a shared resource. Based on this framework, we propose various interesting research directions for extending the results presented in this paper.

Key words: Continuous media – Multimedia databases – Storage systems – Real-time scheduling

Correspondence to: Minos N. Garofalakis

1 Introduction

1.1 Motivation and related work

Next-generation database systems will need to provide support for various forms of multimedia data such as images, video, and audio. Multimedia data types differ from conventional alphanumeric data in their characteristics, and hence require different techniques for their organization and management. Of these new data types, *continuous-media* (CM) data (video and audio) present a major challenge, since they consist of *streams* of media quanta (video frames or audio samples) that must be delivered in a timely fashion. Hence, in contrast to traditional storage managers, a CM storage server needs to ensure that the retrieval and storage of such CM streams proceed at their pre-specified real-time rates.

A CM server, like any other storage server, has a limited amount of resources (e.g., memory, disk/network bandwidth, and disk storage), which places a hard limit on the number of streams that can be simultaneously delivered. Thus, it is a challenging problem to design effective resource management algorithms that can provide on-demand support for a large number of concurrent CM clients. For many application environments, like movies-on-demand, a CM server typically needs to sustain levels of concurrency that far exceed the limits imposed by available resource capacities. This means that the conventional *random access* (or, *fully interactive*) service model that places resource reservations to allocate independent physical channels for each client cannot possibly provide cost-effective solutions in such environments (Özden et al. 1994, 1995a). As a result, several *data-sharing* techniques have been proposed in the literature for increasing the number of concurrent clients beyond the capacity limitations of available resources:

- *Batching* (Aggarwal et al. 1996a, 1996b; Dan et al. 1994; Shachnai and Yu 1995) allows several clients waiting in the server's queue for the same CM clip to share the same stream.
- *Buffering* (or, *Bridging*) (Kamath et al. 1995; Özden et al. 1995c; Shi and Ghandeharizadeh 1997) uses extra memory buffers in a controlled fashion to allow requests that arrive with a small phase difference with respect to

the start of a stream to fetch their data blocks directly from memory (i.e., with no disk access).

- *Piggybacking* (Golubchik et al. 1996) allows clients to view the same clip at different display speeds, so that, eventually, they can catch up with each other and share the same stream.
- *Enhanced Pay-Per-View (EPPV)* (or, *Periodic*) service (Özden et al. 1994; Garofalakis et al. 1997) assigns each clip a *retrieval period*, typically determined by the clip's popularity. Streams retrieving a clip are initiated periodically at offsets equal to the clip's retrieval period and multiple clients can share the same stream[1].

Work on batching has typically concentrated on different disciplines for scheduling requests from the server's queue. The goal is to strike a balance between: (1) fairness, (2) minimizing average waiting time, and (3) minimizing client reneging probability (i.e., the probability that a client's request is cancelled due to excessive waiting). Scheduling disciplines like FCFS, Maximum Queue Length (MQL), and Maximum Factored Queue Length (MFQ) have been proposed and evaluated using simulation models (Aggarwal et al. 1996a; Dan et al. 1994). The main problem with these batching policies is that they are *unpredictable*, in the sense that they cannot offer a guaranteed upper bound on how long a client request must wait in the queue. Furthermore, results on the behavior of such policies (either analytical or by simulation) are typically based on specific probabilistic models of "customer-reneging behavior" whose accuracy is often questionable in practice. Buffering is based on the idea that it is possible to trade extra memory for reduced bandwidth demand. This is a very general approach that is orthogonal to other data-sharing schemes and, consequently, can be incorporated into batching, piggybacking, or EPPV, as an additional optimization (e.g., to facilitate VCR functionality). However, we should note that with current hardware pricing and stream parameters, trading memory for disk bandwidth is often a losing proposition (Leung et al. 1997). Piggybacking uses the fact that small differences in the display rates (e.g., deviations of at most 5% from the normal display rate) are not perceived by the viewer. Nevertheless, the idea raises difficult implementation issues when MPEG-like compression is used, because of inter-frame dependencies.

Compared to other data-sharing schemes (most notably batching), EPPV service offers the advantage of *predictability* – the response time for transmission of a clip to a client is bounded by the clip's retrieval period[2]. This retrieval period is typically determined by the service provider, based on factors such as clip popularity, legal/financial constraints on the distribution rights of a clip, and specific programming choices. Because of the regular pattern of clip retrievals, clients can be informed of the exact time that a specific transmission will start. Thus, even when resources are scarce, the EPPV service model can guarantee *predictable* response times for all incoming requests. An additional benefit of the regularity of EPPV service is that it can also support user interaction through VCR-like operations (Almeroth and Ammar 1996; Özden et al. 1996b). From the service provider's perspective, a desirable feature the regular structure of EPPV service is that it simplifies the periodic scheduling of live events, such as news and sports events, into the program.

Because of the advantages outlined above and its potential to provide scalable, cost-effective CM offerings, EPPV is quickly becoming the service model of choice for telecom, cable, broadcast, and content companies (e.g., PRECEPT Software). 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 (Liu and Layland 1973). The high disk latencies complicate effective utilization of disk bandwidth and storage with reasonable amounts of buffer space, which is an important cost factor in CM 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.

Despite the importance of the resource-scheduling problem for EPPV service, prior work has typically concentrated on other issues such as data layout schemes to efficiently support periodic retrieval (Özden et al. 1994, 1995a) and support for VCR functionality under EPPV (Almeroth and Ammar 1996; Leung et al. 1997). Abram-Profeta and Shin (1997) used simple queuing models to solve the problem of assigning optimal retrieval periods to the set of clips stored at an EPPV server. Their models assume that all clips have the *same length and display rate requirement* and ignore multi-disk data organization issues. None of the above efforts has considered the general problem of EPPV resource-scheduling and, consequently, they can be viewed as orthogonal to our work. A restricted form of the problem that assumes all clips to have *identical* display rates and similar lengths was addressed in the work of Özden et al. (1997).

1.2 Our contributions

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 CM data with (possibly) different display rates (depending on the media type and/or the compression scheme), different periods (depending on the popularity of a clip), 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. Despite its conceptual simplicity, clustered placement can

[1] Depending on the underlying networking technology, clips can be delivered to clients via uni, multi, or broadcast channels.

[2] In existing television terminology, the term "Pay-Per-View" refers to both a prescheduled playback program and a certain pricing mechanism. We refer to our service model as EPPV to emphasize the scheduling aspect of the service without constraining its pricing mechanism.

suffer from disk bandwidth and storage fragmentation, leading to underutilization of available resources.

– **Striping.** Each clip's data is declustered over all available disks. Striping schemes eliminate disk storage fragmentation. On the other hand, as we will see, striping significantly increases the complexity of scalable and cost-effective EPPV services.

In each case, our main objective is to maximize the amount of disk bandwidth that is effectively scheduled under the given layout and storage constraints. This is typically the situation facing large-scale CM servers that periodically need to re-schedule their offerings to adapt to a changing audience, content, and popularity profile[3](Little and Venkatesh 1995; PRECEPT Software 1998). For the clustering scheme, we formulate these optimization problems as generalized variants of the 0/1 knapsack problem (Ibarra and Kim 1975; Lawler 1979; Sahni 1975). Since the problems are clearly $\mathcal{NP}$-hard, we present provably near-optimal heuristics with low polynomial-time complexity. We then present two alternative schemes for striping clips. *Fine-grained striping* (FGS) views the entire disk array as a single large disk in a manner similar to the RAID-3 data organization (Özden et al. 1995d; Patterson et al. 1998). This scheme is conceptually simple and significantly reduces the complexity of the relevant resource-scheduling problems. Nevertheless, FGS suffers from increased disk latency overheads that render it impractical, especially for large disk arrays. *Coarse-grained striping (CGS)* 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 FGS. 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 the EPPV service model under the CGS data layout are non-trivial generalizations of the periodic maintenance scheduling problem (PMSP) (Wei and Liu 1983). Given that PMSP is known to be $\mathcal{NP}$-complete in the strong sense (Baruah et al. 1990), we propose novel heuristic algorithms for scheduling the periodic retrieval of coarse-grained 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 definitions and algorithms to handle the more complex problems introduced by the periodic retrieval under CGS. Thus, our work also contributes to the area of hard real-time scheduling theory by proposing the scheduling tree structure and algorithms as a new approach to periodic maintenance. We also present a scheme for packing multiple scheduling trees to effectively utilize disk bandwidth and storage, and show that all the scheduling problems examined in this paper can be seen as special cases of this general packing formulation. Finally, we present a set of preliminary experimental results that compare the average performance of the schemes proposed in this paper and confirm the superiority of the "CGS + Scheduling Trees" combination under different workloads. We believe that our proposed scheduling tree framework is a powerful tool, applicable to any scenario involving regular, periodic use of a shared resource. A prototype implementation of the framework is currently being built on top of the $\mathcal{F}$*ellini* multimedia storage server at Bell Labs (Martin et al. 1996).

[3] A related problem is *capacity planning.* Given a fixed user profile, determine the minimum system configuration that can accommodate it. This problem is essentially dual to ours, and we believe that most of our techniques are applicable. We will not address capacity planning further in this paper.

1.3 Organization

The remainder of this paper is organized as follows. In Sect. 2, we present our server model architecture and notational conventions. Section 3 describes our resource-scheduling framework and algorithms for clustered CM data layout. FGS and CGS are discussed in Sects. 4 and 5, respectively. In Sects. 6 and 7, we introduce the scheduling tree structure and develop algorithms for the PMSP and the more complex scheduling problems introduced by the EPPV service model under CGS. The experimental evaluation of the average-case behavior of our algorithms is given in Sect. 8. Section 9 discusses some interesting extensions to the schemes developed in this paper. Finally, Sect. 10 concludes the paper and identifies directions for future research. Proofs of theoretical results presented in this paper can be found in the Appendix.

2 Notation and system model

In this section, we present a brief overview of our CM server model and the notation that will be used throughout this paper. First, we present a model of *round-based retrieval* for CM. Our model follows the conventions used in most earlier work on multimedia storage servers (Chen et al. 1993; Gemmell et al. 1995; Özden et al. 1995b; Rangan and Vin 1991, 1993). Next, we present the three different multi-disk data organization schemes considered in this paper (Clustering, FGS and CGS) and describe how the round-based retrieval model adapts to each different organization scenario. Finally, we describe the *matrix-based allocation scheme* (Özden et al. 1996b, 1996c). that tries to optimize the data layout of a clip based on the knowledge of its retrieval period. Table 1 summarizes the notation used in this paper with a brief description of its semantics. Additional notation will be introduced when necessary.

2.1 Retrieving CM data

Consider a single magnetic disk storing a collection of CM clips. 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

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 s)
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 CM server
r_{disk}	Disk transfer rate
c_{disk}	Disk storage capacity
t_{seek}	Disk seek time
t_{lat}	Disk latency

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 into the buffer. This ensures that each stream will have sufficient data to display the corresponding clip continuously. The quantity $T \cdot r_i$ is termed the *retrieval unit* of clip C_i.

During a round, for streams *stream*(C_1), ..., *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 (Silberschatz and Galvin 1994). C-SCAN scheduling ensures that the disk heads move in a single direction when servicing streams during a round. As a result, random seeks to arbitrary locations 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 (Chen et al. 1993; Gemmell et al. 1995; Özden et al. 1995d; Rangan and Vin 1991, 1993). More formally, we require the following inequality to hold[4]:

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

2.2 Multi-disk data organization schemes

2.2.1 Clustering

In clustering, each disk of a multi-disk system is viewed as an autonomous unit – entire clips are stored on and retrieved from a single disk and multiple clips can be clustered on each disk. For our round-based model of data retrieval, this means that Eq. 1 must be satisfied on each disk, where the summation is taken over all streams retrieving clips C_i stored on that disk. Note that by the definition of EPPV service, the number of concurrent streams retrieving clip C_i is exactly $\left\lceil \frac{l_i}{T_i} \right\rceil$. Furthermore, each magnetic disk has a limited storage capacity that cannot be exceeded by the set of stored clips. Thus, if we let $\{C_i\}$ denote the set of clips clustered on any single disk in the server, we require the following conditions to be satisfied:

$$\begin{array}{l} \left\lceil \frac{l_i}{T_i} \right\rceil \cdot \left[\sum_{\{C_i\}} (\frac{T \cdot r_i}{r_{disk}} + t_{lat}) \right] \leq T - 2 \cdot t_{seek} \\ \sum_{\{C_i\}} l_i \cdot r_i \leq c_{disk}. \end{array} \qquad (2)$$

2.2.2 Fine-grained striping

A major deficiency of clustered data organization for large-scale CM services is that it can lead to bandwidth *and* storage fragmentation, and, consequently, underutilization of server resources. Striping schemes eliminate storage fragmentation by declustering a clip's data across all available disks. This essentially means that we no longer need to satisfy a storage capacity constraint on each disk, as long as the total storage requirements of the clips to be scheduled do not exceed the storage capacity of the disk array.

In FGS (Özden et al. 1996a), each retrieval unit of a clip is striped across all n_{disk} disks of the server. Consequently, every stream retrieval involves all the n_{disk} disk heads working in parallel, with each disk being responsible for fetching $\frac{T \cdot r_i}{n_{disk}}$ bits of C_i in each round. This striping strategy is employed in the RAID-3 data distribution scheme (Patterson et al. 1998). For EPPV service, the number of concurrent streams retrieving clip C_i from the array is exactly $\left\lceil \frac{l_i}{T_i} \right\rceil$. Thus, to ensure continuous delivery, the following condition must be satisfied:

$$\left\lceil \frac{l_i}{T_i} \right\rceil \cdot \left[\sum_{i=1}^{N} \frac{T \cdot r_i}{r_{disk} \cdot n_{disk}} + N \cdot t_{lat} \right] \leq T - 2 \cdot t_{seek}, \qquad (3)$$

where N is the total number of clips on the server (see Table 1). Despite its conceptual simplicity, FGS can lead to underutilization of available disk bandwidth due to increased latency overheads (Özden et al. 1996a). This is clearly demonstrated in the above condition for continuity, which shows that, during each round, all disks incur a penalty of t_{lat} for each clip stored *in the entire server*. These latency penalties limit the scalability of an EPPV server based on FGS, since the problem is obviously exacerbated as the size of the disk array grows.

2.2.3 Coarse-grained striping

In the CGS scheme (Özden et al. 1996a), the retrieval units of each clip are mapped to individual disks in a round-robin manner. Consequently, the retrieval of data for a stream on clip C_i proceeds in a round-robin fashion along the disk array. During each round, a single disk is used to fetch a retrieval unit of C_i and consecutive rounds employ consecutive disks[5]. This striping strategy is employed in the RAID-5 data distribution scheme (Patterson et al. 1998).

CGS avoids the large latency overheads of the fine-grained scheme and, consequently, can offer much better

[4] Although our schemes can be extended to handle disk calibration and multi-zone disks (Özden et al. 1995b), these issues are not addressed in this paper.

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

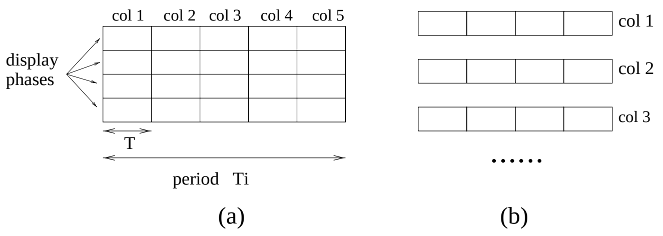

Fig. 1a. A clip matrix. **b** Its layout on disk

scalability and bandwidth utilization (Özden et al. 1996a). On the other hand, supporting periodic stream retrieval requires much more sophisticated scheduling methods than either clustering or FGS. This is because, unlike the two previous data organization schemes, CGS does not impose a steady load on each disk *during each round*. Consider the retrieval of clip C_i from a particular disk in the array. By virtue of the round-robin placement, each stream retrieving data of C_i needs to fetch a retrieval unit of $T \cdot r_i$ bits from that disk periodically, at intervals of n_{disk} rounds. Furthermore, to support EPPV service, the streams retrieving clip C_i are themselves periodic with a period $T_i = n_i \cdot T$. Thus, supporting continuous, periodic service under CGS gives rise to complex *periodic real-time task-scheduling problems* (Liu and Layland 1973) that cannot be reduced to simple algebraic conditions like Eqs. 2 and 3. A concise description of these problems and our proposed solution will be given later in the paper.

2.3 Reducing disk latencies: matrix-based allocation

By definition, the EPPV service model associates with each clip C_i a retrieval period T_i. 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 be more than a few seconds. The *matrix-based allocation scheme* (Özden et al. 1994, 1996c), increases the number of clients that can be serviced under the EPPV model by laying out data based on the knowledge of retrieval periods. The basic idea is to distribute the starting points for the *concurrent display phases* of a clip uniformly across its length, and layout the data that these display phases will need to retrieve during the same round *contiguously* on disk. Each display phase corresponds to a different stream servicing (possibly) multiple clients. This layout method reduces the disk latency overhead incurred during data retrieval and, therefore, increases the amount of disk bandwidth utilized effectively.

More formally, for each clip C_i, the maximum number of concurrent display phases is $\left\lceil \frac{l_i}{T_i} \right\rceil$. Conceptually, clip C_i is viewed as a matrix consisting of elements of length T (Fig. 1a). 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 $\left\lceil \frac{l_i}{T_i} \right\rceil$ 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 $d_i = \left\lceil \frac{l_i}{T_i} \right\rceil \cdot T \cdot r_i$. (Although some columns may actually contain less data than d_i (Özden et al. 1996c), in this paper, we are ignoring possible optimizations for smaller columns.)

To support periodic retrieval, a clip matrix is stored in column-major form (i.e., data in each column is stored contiguously on disk) and its retrieval is performed *in columns* (i.e., one column per round) with each element handed to a different display phase (Fig. 1b). 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 $\left\lceil \frac{l_i}{T_i} \right\rceil$ streams of C_i, rather than $\left\lceil \frac{l_i}{T_i} \right\rceil \cdot t_{lat}$ (using Eqs. 1, 2 and 3). This means that a single disk using the matrix-based scheme can support the periodic retrieval of $C_1, \ldots, C_k$ provided that the following inequality holds:

$$\sum_{\{C_i\}} \left(\frac{d_i}{r_{disk}} + t_{lat}\right) + 2 \cdot t_{seek} \leq T. \qquad (4)$$

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, $\left\lceil \frac{l_i}{T_i} \right\rceil \cdot r_i$.

The matrix-based allocation scheme benefits all three multi-disk data organization strategies considered in this paper by significantly reducing the latency overhead in each round. For example, in Eqs. 2,3 for clustering and FGS the term t_{lat} is replaced by $t_{lat} \cdot \left\lceil \frac{l_i}{T_i} \right\rceil^{-1}$. Thus, we present our results assuming that matrix-based allocation is used. For FGS (CGS), this means that the retrieval unit striped (resp. distributed) across the disks of the server is an entire *column* of C_i rather than a single matrix element. However, we should stress that the schemes presented in this paper are equally applicable to the original data organization methods without the matrix-based scheme optimization.

3 EPPV under clustering

Under a clustered data organization, 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, Eqs. 2 need to hold for each disk. We address this scheduling problem in two stages. First, we present a solution that considers only the bandwidth requirements of clips (essentially, assuming that each disk has infinite storage capacity). Next, we extend our approach to handle disk storage limitations. We present the first case separately, since our results for this case will also prove useful for FGS.

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, equiva-

```
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 $\sum_{C_i \in C'}$ 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
     load($B_j$) = 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 load($B_j$) +
          size($C_i$) $\leq 1$.
     2.2. Set load($B_j$) = load($B_j$) + size($C_i$), value($B_j$) =
          value($B_j$)+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 corresponding to the
     $n_{disk}$ largest 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.)
```

Fig. 2. Algorithm PackClips

lently, its (normalized) disk bandwidth consumption (see Eq. 4); and,

– A *value*: value$(C_i) = \left\lceil \frac{l_i}{T_i} \right\rceil \cdot r_i$, 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* $\{$size(C_i) : $C_i \in C'\}$ *in n_{disk} unit capacity bins such that the total value* $\sum_{C_i \in C'}$ value(C_i) *is maximized.* This problem is a generalization of the traditional 0/1 knapsack optimization problem (Ibarra and Kim 1975; Lawler 1979; Sahni 1975). Thus, it is clearly $\mathcal{NP}$-hard [6]. 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 (Garey and Johnson 1979) with a first-fit packing rule. Briefly, the main idea is to define the value density of clip C_i as the ratio $p_i = \frac{\text{value}(C_i)}{\text{size}(C_i)}$ and pack the clips in decreasing order of density into unit capacity bins using a First-Fit rule. The schedulable subset (and the corresponding schedule) is determined by selecting the n_{disk} "most valuable" bins from the final packing. Algorithm PackClips is depicted in Fig. 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 the disks. The idea is to define the size of a clip C_i as a *2-dimensional size vector* $\mathbf{s}_i = [\text{size}_1(C_i), \text{size}_2(C_i)]$, 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,

$$\text{size}_1(C_i) = \frac{\frac{d_i}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}} \quad \text{and} \quad \text{size}_2(C_i) = \frac{l_i \cdot r_i}{c_{disk}}.$$

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 $p_i = \frac{\text{value}(C_i)}{l(\mathbf{s}_i)}$. The 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 $l(\text{load}(B_j) + \mathbf{s}_i) \leq 1$. 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, 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 $\frac{V_H}{V_{OPT}} \geq \frac{1}{4}$.

We have already noted that clustering can lead to disk storage and bandwidth fragmentation, and this is clearly demonstrated in the rather discouraging worst case bound of Theorem 3.2 – for "bad" lists of clips, PackClips may be able to utilize only as little as one fourth of the raw server capacity. Since storage fragmentation is not an issue when striping is used, 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 $\sum_i l_i \cdot r_i \leq n_{disk} \cdot c_{disk}$ in the description of our striping-based schemes.

4 EPPV under FGS

In the FGS scheme, each column of the clip matrix is declustered across all n_{disk} disks of the server (Fig. 3a). This implies that each clip being retrieved imposes a constant load per round on *all* disks in the server, since each disk is responsible for retrieving $\frac{1}{n_{disk}}$ of the clip's column in each round. Thus, the following condition must be satisfied on each disk:

$$\sum_{i=1}^{N} \frac{d_i}{r_{disk} \cdot n_{disk}} + N \cdot t_{lat} \leq T - 2 \cdot t_{seek}. \qquad (5)$$

To ensure continuous retrieval, all disks in the system must satisfy the same condition (namely, Eq. 5). Conse-

[6] Note that the traditional knapsack problem remains $\mathcal{NP}$-hard even if the size of each item is equal to its value (i.e., the SUBSET SUM problem (Garey and Johnson 1979)) Thus, the fact that size$_1(C_i)$ and value(C_i) are correlated does not affect the hardness of the problem.

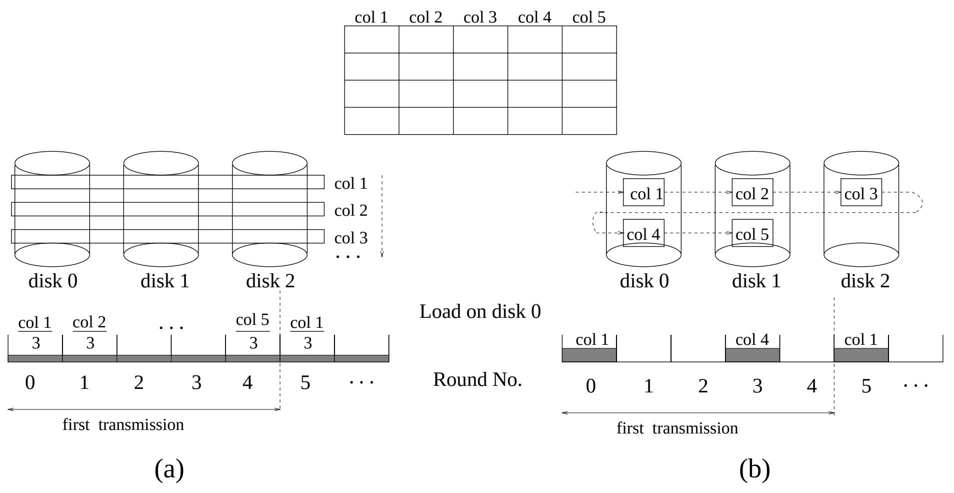

Fig. 3a. Fine-grained striping. **b** Coarse-grained striping

quently, the problem of maximizing the effectively scheduled bandwidth clips under FGS corresponds to a traditional, single-bin, 0/1 knapsack problem with (one-dimensional) clip sizes size$(C_i) = \frac{\frac{d_i}{r_{disk}\cdot n_{disk}}+t_{lat}}{T-2\cdot t_{seek}}$ (from Eq. 5), and values value$(C_i) = \left\lceil \frac{l_i}{T_i} \right\rceil \cdot r_i$ (as in Sect. 3). This is clearly a much simpler version of the knapsack model developed for clustering and traditional knapsack heuristics can be used to provide near-optimal solutions (Ibarra and Kim 1975; Lawler 1979; Sahni 1975). In fact, our PackClips algorithm (with number of bins/disks equal to 1) readily provides a 1/2-approximate heuristic for the problem.

We should once again stress that, despite its conceptual and algorithmic simplicity, FGS suffers from excessive disk latency overheads that severely limit the scalability of the scheme. This fact is analytically shown inEq. 5 and clearly indicated in our experimental results.

5 EPPV under CGS

In the CGS scheme, the columns of a clip matrix are mapped to (and, retrieved from) individual disks in a round-robin manner (Fig. 3b). 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 transmission of C_i, a column of C_i must be retrieved from that disk periodically, at intervals of n_{disk} rounds. From Formula 4, 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* (Liu and Layland 1973) 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 Fig. 3b. 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. Before addressing the scheduling problems associated with this general model of periodic tasks, we briefly review two special cases that essentially correspond to the best and worst case workloads for CGS.

– $\underline{c_i = n_i =}$**multiple**$\underline{(n_{disk})}$ **for all** i**.** In this case, the retrieval of C_i from a particular disk corresponds to a periodic real-time task with period $n_{disk} \cdot T$ and computation time $\frac{d_i}{r_{disk}} + t_{lat}$. Maximizing the effectively scheduled bandwidth can again be formulated as a generalized knapsack problem which, in fact, is identical to the problem defined in Sect. 3.1, but with a slightly different interpretation of terms: the n_{disk} unit-capacity bins now correspond to rounds of length T and the items correspond to retrievals of clip columns. The main advantage of CGS over clustering in this case, is its ability to equally distribute the bandwidth and storage load across all disks. Compared to FGS, the main advantage of CGS is the reduced latency penalty for each clip, which implies better scalability. In fact, a simple analysis shows that, for this special case, CGS is guaranteed to outperform FGS for large disk arrays.
It is important to note that it is not always feasible to reduce the general scheduling problem to this special case, e.g., by "padding" clip lengths or periods so that the equality $c_i = n_i =$multiple(n_{disk}) is satisfied. For example, if the length of a clip is significantly smaller than its period, then $c_i \ll n_i$ and padding the clip to the length of its period is clearly not an effective solution. For example, consider a clip with r_i = 1.5 Mbps, l_i = 5 min, T_i = 100 min, and a system with T = 1 s and n_{disk} = 10. Padding the clip's length to reach its period implies that 1 GB of storage is wasted per clip.

– gcd(n_i, n_j) = 1 **and** $c_i \geq n_{disk}$ **for all** $i, j \neq i$.[7] In this case, using the *Chinese Remainder Theorem* (Knuth 1981), we can prove the following lemma.

Lemma 5.1. Assume that CGS is used for a collection of clips such that gcd(n_i, n_j) = 1 for all $i \neq j$ and $c_i \geq n_{disk}$ for all i. Then, during any time interval of length $n_1 \cdots n_N \cdot T$, there exists a round at which the retrievals for *all* clips collide.

Thus, retrieval periods that are pairwise relatively prime correspond to a worst case scenario for CGS. That is, there exists a round in which $\frac{d_i}{r_{disk}} + t_{lat}$ units of time need to be allocated for each clip C_i. The existence of such worst case collisions means that CGS has to be overly conservative and is typically outperformed by FGS.

Consider the case of arbitrary retrieval periods. Using the *Generalized* Chinese Remainder Theorem (Knuth 1981), we can show the following lemma.

Lemma 5.2. Consider two clips C_1 and C_2, and let $\alpha_i = \min\{ \left\lceil \frac{c_i}{n_{disk}} \right\rceil, \frac{\text{gcd}(n_1,n_2)}{\text{gcd}(n_1,n_2,n_{disk})} \}$, $i = 1, 2$. 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 5.2 identifies a necessary and sufficient condition for the *collision-free* scheduling (or, *mergeability* (Yu et al. 1989) of two clip retrieval patterns. Our result extends the result of Yu et al. (1989) on merging two simple periodic patterns to the case of periodic tasks consisting of equidistant subtasks. Furthermore, Lemma 5.2 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 $i \neq j$. (Note that, for two clips, this condition is obviously true with k = gcd(n_1, n_2).)

Lemma 5.3. 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 $\alpha_i = \min\{ \left\lceil \frac{c_i}{n_{disk}} \right\rceil, \frac{k}{\text{gcd}(k,n_{disk})} \}$. The retrieval of C can be scheduled without collisions *if and only if* $\sum_{i=1}^{N} \alpha_i \leq k$.

Unfortunately, Lemma 5.2 cannot be extended to the general case of multiple clips with arbitrary periods. In fact, in Sect. 6, 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 5.2 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 5.2 (extended for three clips) fails to hold, since gcd(n_1, n_2, n_3) = 2 < $\sum_{i=1}^{3} \alpha_i$ = 3.

[7] The gcd() function returns the *greatest common divisor* of a set of integers.

6 The scheduling tree structure

In the previous section, we identified the scheduling problem that arises when supporting EPPV service under CGS and examined some special cases. In this section, we address the general problem. 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* (Baruah et al. 1990; Wei and Liu 1983), 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 Sect. 5.

6.1 Periodic maintenance scheduling

The *k-server periodic maintenance scheduling problem (k-PMSP)* (Baruah et al. 1990) 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. (1990) 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[8].

Definition 6.1. 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 $\text{ancestor\_node}_j(n)$ represent the weight of the ancestor node of

[8] 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 (Stankovic and Ramamritham 1993).

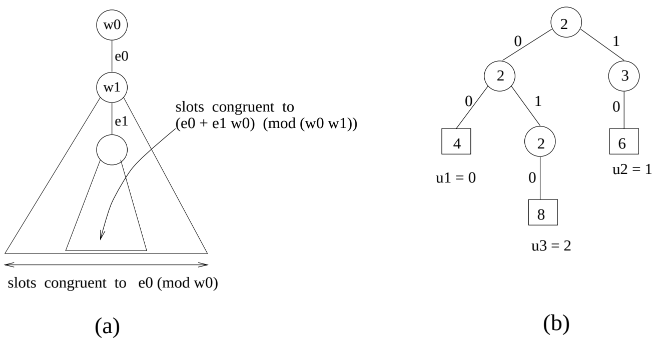

Fig. 4a. The scheduling tree structure. **b** A tree for the set of tasks in Example 1

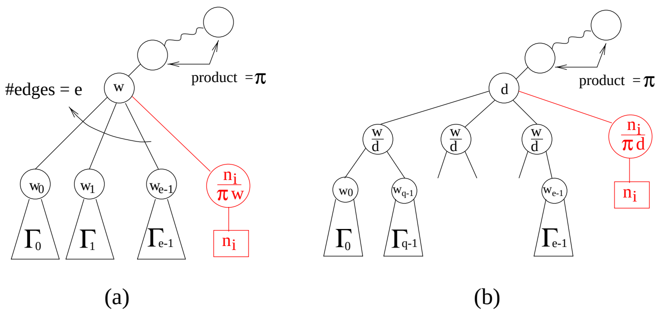

Fig. 5a. Placing a period n_i under a scheduling tree node without splitting. **b** Period placement when the node is split

n at level j, and let $\text{ancestor\_edge}_j(n)$ denote the weight of the ancestor edge of n at level j, where $j \le level(n)$. Finally, define $\pi_j(n) = \prod_{i=0}^{j} \text{ancestor\_node}_i(n)$ for $0 \le j \le level(n)$.

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 from the scheduling tree structure as follows:

$$u_i = \text{ancestor\_edge}_1(n_i) + \sum_{j=2}^{l} \text{ancestor\_edge}_j(n_i) \cdot \pi_{j-2}(n_i). \quad (6)$$

Note that, by the definition of the scheduling tree structure, the set of node weights on a path from the root to a leaf (i.e., period) node represents a *factorization* of that period. The edge weights along the root-to-leaf path intuitively identify the slot number moduli that the leaf node "occupies" in the periodic schedule. Some additional intuition for the scheduling tree structure and the above formula is provided in Fig. 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.

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 6.2. Let $\Gamma_1, \ldots, \Gamma_k$ be scheduling trees for $P_1, \ldots, P_k$, where $P_1, \ldots, P_k$ is a partitioning of of the periods in P. The trees Γ_i and Γ_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 partition of P.

Corollary 6.1. 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 6.3. An *internal* node n at level l is *candidate for period* n_i if and only if $\pi_{l-1}(n) | n_i$ and $\gcd(w(n), \frac{n_i}{\pi_{l-1}(n)}) \ge \frac{w(n)}{w(n)-e(n)}$.

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

- <u>**If** $\pi_l(n) | n_i$</u> then the condition in Definition 6.3 guarantees that n has at least one free edge at which n_i can be placed (Fig. 5a).
- <u>**If** $\pi_l(n) \not| n_i$</u> 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 Fig. 5b; 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, the condition in Definition 6.3 guarantees that the newly created parent node will have 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 Sect. 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

```
Algorithm BUILDTREE(C, value)

Input: A set of UET 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 $\sum_{C_i \in C'}$ 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 value($C_i$) $\geq$ 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 loss($n$) = $\{C_j \in L - \{C_i\} \mid cand(\Gamma \cup \{n_i\}_n) = \emptyset\}$ and value(loss($n$)) = $\sum_{C_j \in \text{loss}(n)}$ value($C_j$).
   2.3. Place $n_i$ under the candidate node $M$ such that value(loss($M$)) = $\min_{n \in cand(n_i, \Gamma)}$\{value(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 - $ loss($M$).
   2.5. For each task $C_j \in L$, update the candidate node set $cand(n_i, \Gamma)$.
```

Fig. 6. Algorithm BUILDTREE

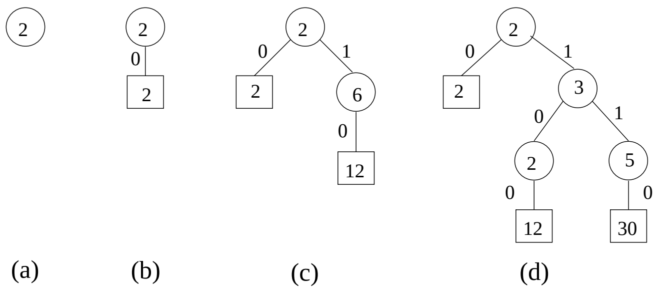

Fig. 7a–d. Construction of a scheduling tree for the set of tasks in Example 2

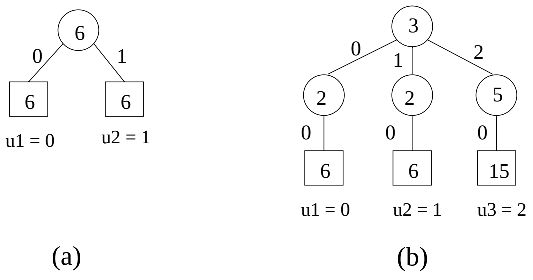

Fig. 8a,b. Illustration of the new splitting rule

using the postorder node numbers (i.e., left-to-right order). When a period is scheduled in Γ, 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 Γ). Algorithm BUILDTREE is depicted in Fig. 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 Fig. 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 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.

6.2 Scheduling equidistant subtasks

In Sect. 5, we identified a clip retrieval under CGS 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 Eq. 6) 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 Eq. 6.

Example 3. Figure 8a illustrates a scheduling tree with two tasks with periods $n_1 = 6$, $n_2 = 6$ assigned to slots 0 and 1. Figure 8b 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.

In this setting, the notion of a candidate node is defined as follows.

Definition 6.4. 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 $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$.

```
Algorithm FREESLOT($n, n_i, u$)

Input: A scheduling tree node $n$, a period $n_i$, and a specific time slot $u$.
Output: TRUE iff $n_i$ can be scheduled in slot $u$ under node $n$; FALSE otherwise.

1. If $n$ is not a candidate for $n_i$ then return(FALSE).
2. Let $l = level(n)$ and let $t = \text{ancestor\_edge}_1(n) + \sum_{j=2}^{l} \text{ancestor\_edge}_j(n) \cdot \pi_{j-2}(n)$.
3. If $\pi_{l-1}(n)$ does not divide $u - t$ then return(FALSE);
   Else let $u = \frac{u-t}{\pi_{l-1}(n)}$.
4. Let $d = \gcd(n_i, w(n))$ and $e = u \bmod d$.
5. If all edges of $n$ labeled $k \cdot d + e$, for $k = 0, \ldots, \frac{w(n)}{d} - 1$ are free
   then return(TRUE);
   Else return(FALSE).
```

Fig. 9. Algorithm FREESLOT

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 6.4, and use a simple predicate based on Eq. 6, 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. Algorithm FREESLOT, depicted in Fig. 9, is a predicate for checking the availability of a specific time slot u under a candidate node n for period n_i. The correctness of the algorithm follows from the fact that Eq. 6 can be rewritten in nested form as follows:

$$\begin{aligned} u_i = \; & \text{ancestor\_edge}_1(n_i) + \text{ancestor\_node}_0(n_i) \cdot \\ & (\text{ancestor\_edge}_2(n_i) + \text{ancestor\_node}_1(n_i) \cdot \\ & (\ldots (\text{ancestor\_edge}_{l-1}(n_i) + \text{ancestor\_node}_{l-2}(n_i) \cdot \\ & \text{ancestor\_edge}_l(n_i)) \ldots) \,. \end{aligned}$$

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 min). 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 Eq. 6, 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 max 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). Based on the above scheme, we can define the notion of a candidate node as follows.

Definition 6.5. Let n be an internal node at level l. Let n_i be a period and define $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$. Node n is *candidate for period* n_i if and only if $\pi_{l-1}(n) | n_i$ and the following conditions hold

1. If n is the root node, n has a free edge.
2. If $level(n) = 1$, there exists an $i \in \{0, \ldots, d-1\}$ such that all (non-disabled) edges of n whose sum of weight plus distance is congruent to $(i + j) \pmod d$, for $0 \leq j < s_i$, are free.
3. If $level(n) \geq 2$,
   - 3.1 there exists an $i \in \{0, \ldots, d-1\}$ such that all edges of n with weight congruent to $i \pmod d$ are free; and,
   - 3.2 $s_i - 1 + \text{ancestor\_edge}_2(n) < \text{ancestor\_node}_1(n)$ and $s_i + \text{ancestor\_edge}_2(n)$ is less than or equal to the weight of the (non-disabled) edge following $\text{ancestor\_edge}_2(n)$, if there is such an edge.

Note that clause 2 ensures that edge distances are maintained when first-level nodes are split. Based on the above definition of candidate nodes, BUILDEQUIDTREE (shown in Fig. 10) can be used to construct a scheduling tree in polynomial time. A proof of the correctness of BUILDEQUIDTREE can be found in the Appendix.

Example 4. Consider three tasks C_1, C_2 and C_3 with $s_1, s_2, s_3 = 2, 1, 3$, $n_1, n_2, n_3 = 12, 18, 10$, and $n_{disk} = 2$. Figure 11

```
Algorithm BUILDEQUIDTREE

Input: A set of 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 $C_i$. Each task consists of subtasks placed at intervals
       of $n_{disk}$.
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 $n_{disk}$.
  2. For each task $C_i$ in $L$ (in that order)
     2.1 Select a candidate node $n$ for $n_i$ in $\Gamma$. (Ties are broken in
         favor of nodes at higher levels).
     2.2 If $w(n) \not| n_i$, split $n$.
     2.3 Schedule the first subtask of $C_i$ under $n$. (Ties are broken in
         favor of edges with smaller weights.)
     2.4 Let $d$ be the distance of the ancestor edge at the first level of
         the leaf corresponding to $n_i$.
         Set the distance of this edge to $\max\{d, s_i - 1\}$.
```

Fig. 10. Algorithm BUILDEQUIDTREE

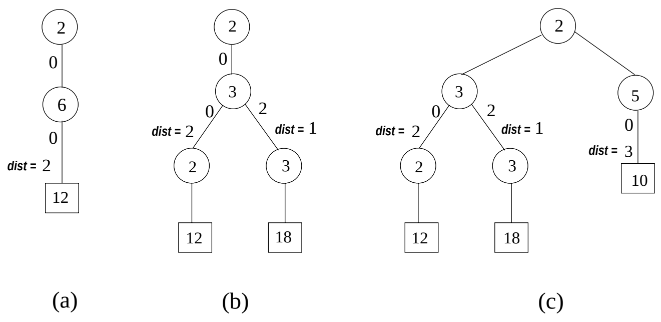

Fig. 11a–c. Scheduling equidistant subtasks with edge disabling

illustrates the three states of the scheduling tree after placing tasks C_1, C_2, and C_3, respectively.

6.3 Handling slots with multi-task capacities

An interesting property of the scheduling tree formulation is that it 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 Sect. 5, this is exactly the case for the rounds of EPPV retrieval under CGS. Using the notation of Sect. 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 6.6. Let n be a leaf node for of a scheduling tree Γ 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 6.7. 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 6.4 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 our BUILDEQUIDTREE algorithm can be used without modification to produce a scheduling tree for the multi-task capacity case.

7 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 Sect. 6.3. 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 6.6. 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 Γ_1 and Γ_2 that are constructed independently. Let e_1 be an edge emanating from the root node n_1 of Γ_1 and e_2 be an edge emanating from the root node n_2 of Γ_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 strategies for combining forests. As an example, consider the following strategy. Assume a collection of independently built scheduling forests $\{F_i\}$. Let d be the gcd of all the root nodes of all the trees in every forest. Let $F_i(j)$, $0 \leq j < d$, denote the collection of all subtrees rooted at a first level node (i.e., a child of the root) in each tree within forest F_i, such that the weight of the edge connecting the subtree to the root is congruent to j (mod d). As previously, we define the size and value of $F_i(j)$ as $\mathsf{size}(F_i(j)) = \max_{n_k \in F_i(j)}\{\mathsf{load}(n_k)\}$, where n_k is any leaf node in $F_i(j)$, and $\mathsf{value}(F_i(j)) = \sum_{C_k \in F_i(j)} \mathsf{value}(C_k)$. Finally, let $F(j)$ denote the collection of all $F_i(j)$'s for a fixed value of j. We consider three different cases of the scheduling problem.

1. No subtasks. In this case, each task is a simple periodic task. We are then faced with a packing problem that can be described as: *Given d collections of objects $F = \{F(0), \ldots, F(d-1)\}$, for each collection $F(j)$, $0 \leq j < d$, determine a subset $F'(j)$ of $F(j)$ and a packing of $F'(j)$ in a unit capacity bin such that the total value $\sum_{F_i(j) \in F'[j]} \mathsf{value}(F_i(j)$ is maximized.* Since each collection an be treated independently, our problem corresponds to a traditional 0/1 knapsack optimization problem. Thus, knapsack heuristics (e.g., algorithm PackClips$(F(j), 1)$) can be used for each collection $F(j)$ of objects. (The set of scheduled tasks is defined by the set of subtrees selected in the packing.)
2. $\mathrm{gcd}(n_{disk}, d) > 1$: In this case, if we set d to be equal to $\mathrm{gcd}(n_{disk}, d)$, then the optimization problem is the same as in case (1). In other words, in spite of the subtasks, we can pack subtrees of different forests, rather than packing the entire forest. This is because, if $\mathrm{gcd}(n_{disk}, d) > 1$ holds, then all the subtasks of a task reside in subtrees rooted at edges (emanating from the root) with weights that are congruent to $j \pmod{\mathrm{gcd}(d, n_{disk})}$ for some j $(0 \leq j < d)$.
3. Otherwise. With each forest F_i, we associate a d-dimensional size vector (with the j^{th} component equal to $\mathsf{size}(F_i(j))$) and a value $\mathsf{value}(F_i) = \sum_{j=1}^{d} \mathsf{value}(F_i(j))$. We are then faced with a d-dimensional variant of our original (i.e., worst case collision) packing problem, in which forests are packed into a d-dimensional unit capacity bin with the objective of maximizing the accumulated value. Again, heuristics (like PackClips) based on d-dimensional vector packing (Coffman et al. 1984) can provide approximate solutions.

8 Experimental performance evaluation

8.1 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 min (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 between 40 and 60 min and a "cold region" with periods between 150 and 180 min.
– **Workload #2** consisted of small video clips with lengths between 2 and 10 min (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 between 20 and 30 min and a "cold region" with periods between 40 and 60 min.

We experimented with each component executing in isolation and with mixed workloads consisting of mixtures of type #1 and type #2 workloads. For each experimental run, an appropriate number of type #1 and/or type #2 video clip requests were generated randomly, using a uniform distribution to select specific values for clip lengths, display rates, and periods from the corresponding ranges. Note that the ranges were actually discretized for our experiments. For example, instead of the continuous range 90–120 we used the set 90, 100, 110, 120. This allowed us not only to model more realistic scenarios for our video server setting (e.g., support only a finite number of periods and display rates), but also to avoid "bad" examples for our periodic scheduling problem (e.g., periods that are relatively prime). We concentrated on scaleup experiments in which the number of generated video clip requests was chosen so that 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 (Sect. 3.2).) Once the set video clip requests in the workload was finalized, the requests were handed to the appropriate scheduling algorithm for each data layout strategy to try to produce a valid schedule given the bandwidth and storage limitations of the server. For clustering and FGS, we used the 2- and 1-dimensional version of PackClips, respectively. For CGS, we employed our scheduling tree building algorithm (to build a collection of trees for the given periodic requests) followed by a simple tree-packing scheme based on the "worst case collision" assumption (described in Sect. 7). In all cases, our basic performance metric was the *effectively scheduled disk bandwidth* (in Mbps); that is, the amount of the server's disk bandwidth that each scheduling scheme was able to utilize effectively for the given workload. (The graphs presented in the next section are indicative of the results obtained over different 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 GB 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 s. As part of our future work, we plan to examine the effect of these parameters on the performance of our scheduling schemes. Table 2 summarizes our experimental parameter settings.

Table 2. Experimental parameter settings

Disk Params.		Workload #1		Workload #2	
r_{disk}	80 Mbps	l_i	90–120 min	l_i	2–10 min
c_{disk}	4 GB	T_i	40–60 , 150–180 min	T_i	20–30, 40–60 min
t_{seek}	24 ms	r_i	1.5 Mbps	r_i	2–4 Mbps
t_{lat}	9.3 ms	No. clips	20–200	No. clips	80–200
T	1 s	Hot clips	5%–50%	Hot clips	5%–50%

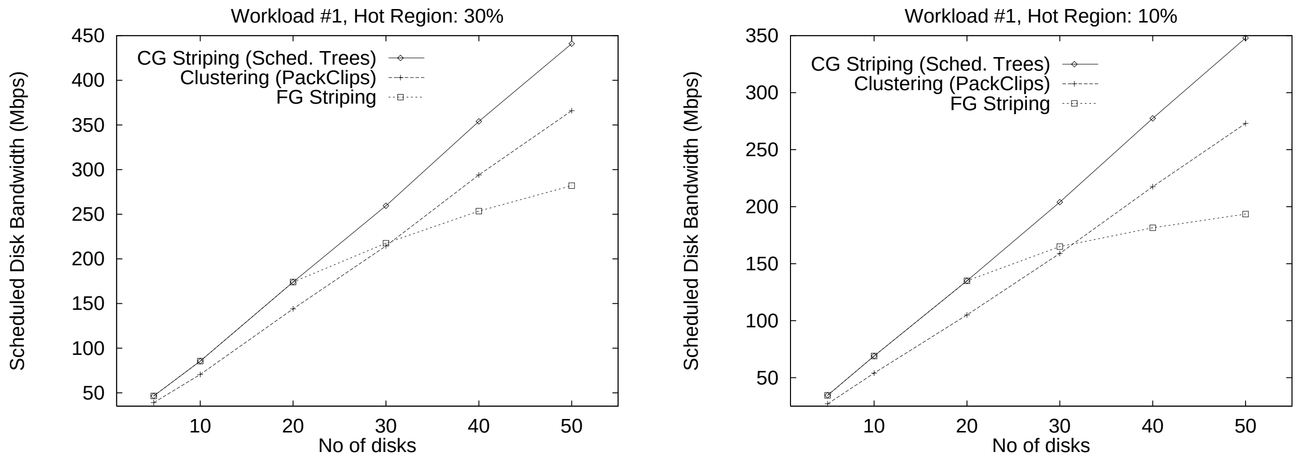

Fig. 12a. Workload #1, 30% hot. **b** Workload #1, 10% hot

8.2 Experimental results

The results of our experiments with type #1 workloads with hot regions of 30% and 10% are shown in Fig. 12a and b, respectively. Clearly, the CGS-based scheme outperforms both clustering and FGS over the entire range of values for the number of disks. Observe that, for type #1 workloads and for the parameter values given in Table 2, 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 Fig. 12. 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 FGS schemes quickly deteriorates as the size of the disk array increases. This confirms our remarks on the limited scalability of FGS in Sect. 4. 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 Fig. 13a. Again, the CGS-based scheme outperforms both Clustering and FGS over the entire range of n_{disk}. Note that, compared to type #1 workloads, the relative performance of clustering and FGS 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 13a clearly demonstrates the devastating effects of this bandwidth wastage and the need for periodic scheduling algorithms.

Finally, Fig. 13b depicts the results obtained for a mixed workload consisting of 30% type #1 clips and 70% type #2 clips. CGS is once again consistently better than FGS 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 FGS. Still, its wastefulness of disk bandwidth for short clips does not allow it to perform at the level of CGS.

9 Extensions

9.1 Improvements for long periods

In general, the period T_i of a clip C_i may be greater than its length l_i. The algorithms presented in Sects. 3 and 4 for clustering and FGS can be used to schedule such clips, however, they may be unnecessarily restrictive. This is because for clustering and FGS, PACKCLIPS reserves disk time equal to $\frac{d_i}{r_{disk}} + t_{lat}$ and $\frac{d_i}{n_{disk} \cdot r_{disk}} + t_{lat}$, respectively, every T units of time for clip C_i. However, if the length the clip

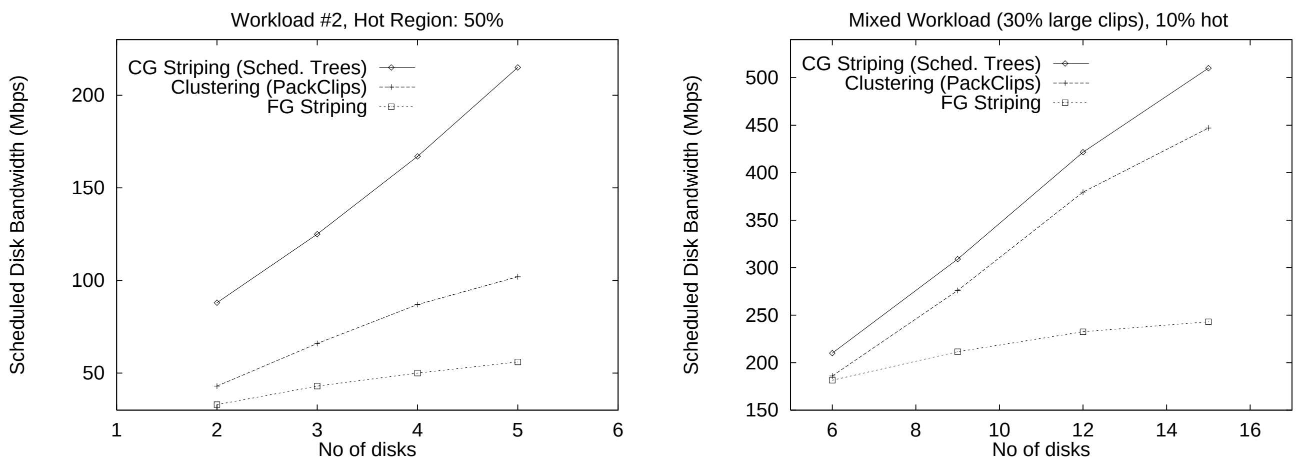

Fig. 13a. Workload #2, 50% hot. **b** Mixed workload (30–70%), 10% hot

is much smaller than its period, then in every T_i time slots the reserved disk capacity in $T_i - l_i$ time slots is wasted. The effects of this bandwidth wastage and the need for periodic scheduling techniques are also demonstrated in our experimental results in Sect. 8.2.

Under clustering and FGS, the retrieval of a clip C_i can be modeled as a collection of periodic real-time tasks with period $T_i = n_i \cdot T$, where each task consists of a collection of c_i subtasks that are T time units apart and have a computation time equal to the column retrieval time. (c_i is the number of columns in C_i.) Note that the only difference between this task model and the one defined in Sect. 5 is that the distance between consecutive subtasks is only one time slot (rather than n_{disk}). Our scheduling tree algorithms and the packing-based schemes for combining forests and trees can easily be modified to deal with this case.

Obviously, to deal with the storage dimension for clustering, the packing-based scheme presented in Sect. 7 needs to become two-dimensional. That is, each forest F_i is characterized by a two-dimensional size vector $s_i = [\mathsf{size}_1(F_i), \mathsf{size}_2(F_i)]$ (similar to the one described in Sect. 3.2), where $\mathsf{size}_1(F_i)$ is the maximum disk bandwidth requirement of any task scheduled within F_i and $\mathsf{size}_2(F_i)$ is the total storage requirement of all the tasks scheduled within F_i. More formally,

$$\mathsf{size}_1(F_i) = \max_{C_j \in F_i}\left\{\frac{\frac{d_j}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}\right\} \quad \text{and}$$

$$\mathsf{size}_2(F_i) = \sum_{C_j \in F_i} \frac{l_j \cdot r_j}{c_{disk}}.$$

Using these definitions, the two-dimensional PACKCLIPS algorithm can provide an approximate solution to the value maximization problem for a given collection of forests. Note that, if the number of subtasks of a task C_i is equal to $n_i = \frac{T_i}{T}$, then C_i will occupy all available time slots in a scheduling tree. Thus, if $T_i > l_i$ holds for each clip C_i, our scheme reduces exactly to the one in Sect. 3.

FGS can be handled in a similar manner. In this case, each forest F_i is characterized by a scalar (i.e.,one-dimensional) size $\mathsf{size}_1(F_i) = \max_{C_j \in F_i}\left\{\frac{\frac{d_j}{n_{disk} \cdot r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}\right\}$, and forests can be packed using one unit capacity bin.

9.2 Conventional data layout

Previously in this paper, we assumed that clips are stored on disks using the matrix-based layout scheme. That is, each column of a clip matrix is stored contiguously. A column is nothing more than the total amount of data that needs to be retrieved in a round for all concurrent display phases. Thus, the matrix-based layout provides the nice property of reducing the disk latency overhead within a round for all the concurrent phases to a single t_{lat}. On the other hand, our scheduling and packing algorithms can also handle conventional data layout schemes that do not exploit the knowledge of retrieval periods during data layout.

Assume the conventional data layout scheme that stores the clip data contiguously on disk (i.e., stores the clip matrix in row-major order). This scheme has been the basis of most work on CM. Let b_i denote the disk bandwidth overhead for supporting the periodic retrieval of clip C_i. If $T_i \leq l_i$, then b_i is the same under both the conventional and the matrix-based scheme. However, if $T_i > l_i$, then clustering and CGS require $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil \left(\frac{T \cdot r_i}{r_{disk}} + t_{lat}\right)$ under conventional layout, and only $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil \frac{T \cdot r_i}{r_{disk}} + t_{lat}$ under matrix-based layout. Similarly, if $T_i > l_i$, then FGS requires $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil \left(\frac{T \cdot r_i}{n_{disk} r_{disk}} + t_{lat}\right)$ under conventional layout, and only $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil \frac{T \cdot r_i}{n_{disk} r_{disk}} + t_{lat}$ under matrix-based layout. Thus, our packing and scheduling algorithms remain the same once b_i and, thus, $\mathsf{size}_1(C_i) = \frac{b_i}{T - 2 \cdot t_{seek}}$ are appropriately redefined.

9.3 Random access service

In addition to supporting EPPV service, our tree-based scheduling algorithms can also offer support for *random access* service models, where resource reservations have to be placed to allocate an independent channel for each admitted client.

Most CM storage systems are built using the round-based disk scheduling/buffer management algorithm described in Sect. 2. That is, data for CM streams is retrieved into a buffer cache from disks in rounds of length T. For each stream C_i, a buffer of size $2 \cdot T \cdot r_i$ is reserved for the duration of the stream and a disk time of length $\frac{T \cdot r_i}{r_{disk}} + t_{lat}$ is reserved in every round. In each round, while the stream is consuming $T \cdot r_i$ bits of data from the buffer cache, the next $T \cdot r_i$ bits of data that the stream will consume in the next round is retrieved from disk into the buffer cache. The optimum value of T that maximizes the throughput depends on the available buffer space, disk bandwidth, latency and rates of the incoming stream requests. In order to maximize the throughput under this basic round-based approach, the value of T (i.e., the length of the round) needs to be changed dynamically depending on the incoming requests. However, changing the value of T dynamically complicates the system design.

An alternative strategy for supporting continuous retrieval of CM data is to prefetch a constant amount, say d bits, for each stream independent of its rate and maintain a fixed round size. The consumption time of d bits depends on the rate r_i of a stream C_i. More specifically, if we denote this time by P_i, then P_i can be estimated as $P_i = \frac{d}{r_i}$. Hence, for each stream C_i, instead of prefetching $T \cdot r_i$ bits in every round, d bits can be prefetched in every $p_i = \frac{P_i}{T}$ rounds. Thus, in every p_i rounds, $\frac{d}{r_{disk}} + t_{lat}$ time must be reserved for a stream C_i (If each clip is striped over n_{disk} disks in a round-robin manner, then d bits need to be retrieved from a disk in every $n_{disk} \cdot p_i$ roundsm and in this case we denote this number of rounds by p_i). Prefetching d bits in every p_i rounds ensures that stream C_i will have sufficient data to display the corresponding clip continuously. This length is independent of a stream's rate and constant for each stream. Moreover, in order to reduce the buffer space requirement of a stream from $2 \cdot d$ to $d + T \cdot r_i$, one needs to schedule each retrieval for C_i exactly p_i units apart. Thus, this approach results in a collection of simple periodic tasks.

In this alternative approach, for each disk (there are n_{disk} disks in the case of CGS or Clustering and one "big" disk in case of FGS), $\frac{T - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{lat}}$ scheduling trees can be maintained. When a new stream C_i arrives, the admission controller can check whether C_i can be inserted into any of the scheduling trees and whether there is $d + T \cdot r_i$ bits of buffer space to reserve for C_i, and if this is the case, stream C_i can be admitted. The scheduling tree ensures that the data retrieval for each stream within the tree will never collide in the same round with the data retrieval for another stream in the tree. On the other hand, the data retrieval for a stream in one tree may collide in a round with the one of another stream in another tree. However, the number of data retrievals that collide in a round will never exceed $\frac{T - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{lat}}$, since only so many trees are maintained for each disk. Since each round has enough capacity for retrieving $\frac{T - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{lat}} \cdot d$ bits from a disk, this approach ensures that each stream C_i will always have sufficient data to display the corresponding clip continuously, while requiring only $d + T \cdot r_i$ buffer space.

Example 5. Consider a single disk system with r_{disk} = 40 Mbps, t_{lat} = 9.3 ms and t_{seek} = 14 ms. Suppose that the value of T is set to 1 s in both approaches. Let d be 1.5 Mb. Suppose that for a while all the incoming requests have a rate of r_1 = 1.5 Mbps. Both schemes will support approximately 21 requests. Now, let us assume after a while all the incoming requests have a rate of r_2 = 28.8 Kbps. If the value of T is not changed in the basic round-based scheme, this scheme can support approximately 99 requests with rate r_2. On the other hand, the scheduling-tree based approach can support 1092 requests with rate r_2.

10 Conclusions and future research directions

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, FGS, and CGS. In each case, the periodic nature of the EPPV service model raises a host of interesting resource-scheduling problems. For clustering and FGS, 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 FGS incurs high disk latency overheads that limit its scalability. CGS, 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 CGS to be a generalization of the periodic maintenance scheduling problem (Wei and Liu 1983) and developed a number of novel concepts and algorithmic solutions to address the issues involved. We also presented a preliminary set of experimental results that verified our expectations about the average performance of the three schemes: clustering can lead to fragmentation and underutilization of resources and the performance of FGS does not scale linearly in the number of disks due to increased latencies. Our novel tree-based algorithm for CGS emerged as the clear winner under a variety of randomly generated workloads. Finally, we demonstrated that the novel resource-scheduling framework developed in this paper is powerful enough to handle a number of different multimedia-related scheduling problems.

The EPPV service model and the scheduling tree framework proposed in this paper suggest several directions for future research.

(1) *Handling task release/departure dates.* The schemes presented in this paper are based on the assumption that the periodic tasks corresponding to the retrieval of clips from the EPPV server are all available to be scheduled at time 0. Furthermore, the tasks are *permanent*, in the sense that, once scheduled, they will not depart from the system (at least, not until the server decides to reschedule its offerings). In real life, however, the clip retrievals will have specific *release* and *departure* dates that limit their lifetime in the playback program. For example, the offering of an R-rated presentation should probably be limited to the evening hours (e.g., 9pm to 5am). A possible solution using the existing methods, would be to construct multiple scheduling trees for different time intervals (based on expected releases/departures) and then constrain the algorithm that combines trees to only

combine trees for the same interval. A more general approach would be to try to directly incorporate the concept of release and departure dates in the scheduling tree structure. The main idea is that the scheduling structure needs some way of accounting for *temporary* periodic tasks that are active only during specific time intervals. This essentially corresponds to an extension of our model of periodic real-time tasks, where the *offering of a clip itself* can be either periodic (e.g., daily 9pm–5am) or one-shot (e.g., a live debate on November 12). Designing scheduling structures and algorithms for this extended task model is a challenging and important problem for future research.
(2) *Incorporating memory buffering.* The schemes presented in this paper assume that: (1) the minimum possible amount of buffering is provided to each active stream; and, (2) the buffering constraint can never be the bottleneck when deciding the subset of the clips to schedule (i.e., there will always be enough buffers to support the clips chosen). When server memory is scarce, assumption (2) may not be valid. In such cases, the buffering constraint can be incorporated in our scheduling framework by viewing buffer requirements as an additional dimension to the "size" of a clip. Our schemes can then be readily extended using a *vector-packing* formulation, as in the 2-dimensional PACKCLIPS algorithm[9]. Note that the buffer constraint can be ignored if the buffer size is at least $2 \cdot T \cdot r_{disk} \cdot n_{disk}$. The situation is much more interesting when assumption (1) is invalid. The existence of additional memory buffers allows the clip retrieval tasks some flexibility, in the sense that data for a stream can be retrieved within a range of several rounds (rather than the round right before its consumption). This flexibility suggests connections to traditional real-time scheduling problems, like the *pinwheel problem* (Chan and Chin 1993) or *distance-constrained task scheduling* (Han et al. 1996). If this additional buffer memory is shared among streams, then a number of interesting issues arise with respect to buffer management policies.
(3) *Better heuristics for building scheduling trees.* Although our experiments have shown that the greedy BUILDTREE heuristic performs very well for sets of periods typically encountered in EPPV practice, we have not been able to obtain theoretical bounds on its worst case performance. Obtaining such bounds is an interesting open problem. Also, it may be reasonable to assume that additional information on the clip retrieval periods is known. For example, the scheduler may have access to the (canonical) *prime factorizations* (Knuth 1981) of all periods. In other situations, the scheduler may be allowed some flexibility in the sense that it can *modify* the given clip retrieval periods within prespecified bounds (e.g., $\pm$ 20%). A natural question that arises is whether it is possible to design better heuristic algorithms that take advantage of such extra information and scheduling flexibilities.
(4) *Incorporating different retrieval strategies.* The simple model of round-based C-SCAN retrieval with a fixed common round length T described in Sect. 2 is just one possible retrieval strategy. Many other strategies that have been proposed in the literature (e.g., the GSS scheme (Chen et al. 1993) that tries to optimize buffer usage) can be readily incorporated into our framework. More complicated problems arise when the differences in the clip display rates are such that *different round lengths* have to be used for different clips in order to use disks and memory effectively. For example, clips with small rates should use large round lengths to reduce the disk bandwidth wasted to latency overheads, whereas clips with large rates should use smaller round lengths to reduce their buffer requirements. Handling different round lengths requires significant and non-trivial extensions to our scheduling tree framework and methodology.
(5) *Dealing with variable bit-rate (VBR) data.* The schemes presented in this paper can deal with VBR data using standard techniques. For example, the server can push data for clip C_i at a constant rate r_i and sufficient buffer space is allocated at the client to smooth variations with respect to r_i. Note that, by the nature of EPPV service, different display phases of the same clip will typically (assuming the clip's period is smaller than its length) overlap in time, and, consequently, share resources such as disk bandwidth and buffer space. This presents possibilities for additional optimizations for VBR data under EPPV, by controling the overlap of concurrent streams on the same clip to reduce their combined resource requirements (e.g., overlapping bandwidth "peaks" in one phase with "valleys" in another). Exploring such optimizations is left as an open issue for future research.

[9] Of course, the same approach can be used for other potential resource bottlenecks, e.g., network bandwidth.

References

1. Abram-Profeta EL, Shin KG (1997) Scheduling Video Programs in Near Video-on-Demand Systems. In: Proceedings of ACM Multimedia, November 1997, Seattle, Wash. ACM Press, New York, NY, pp 359–369
2. Aggarwal CC, Wolf JL, Yu PS (1996a) On Optimal Batching Policies for Video-on-Demand Storage Servers. In: Proceedings of the International Conference on Multimedia Computing and Systems, June 1996, Hiroshima, Japan. IEEE-CS Press, Los Alamitos, CA, pp 253–258
3. Aggarwal CC, Wolf JL, Yu PS (1996b) The Maximum Factor Queue Length Batching Scheme for Video-on-Demand Systems. Technical Report RC 20621 (11/11/96), IBM Research Division, Yorktown Heights, NY
4. Almeroth KC, Ammar MH (1996) On the Use of Multicast Delivery to Provide a Scalable and Interactive Video-on-Demand Service. IEEE J Sel Areas Commun 14(6)
5. Baruah S, Rosier L, Tulchinsky I, Varvel D (1990) The Complexity of Periodic Maintenance. In: Proceedings of the International Computer Symposium, 1990, Taipei, Taiwan, pp 315–320
6. Chan MY, Chin F (1993) Schedulers for Larger Classes of Pinwheel Instances. Algorithmica 9:425–462
7. Chen M-S, Kandlur DD, Yu PS (1993) Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams. In: Proceedings of ACM Multimedia, August 1993, Anaheim, Calif. ACM Press, New York, NY, pp 235–242
8. Coffman EG jr, Garey MR, Johnson DS (1984) Approximation Algorithms for Bin-Packing – An Updated Survey. In: Algorithm Design for Computing System Design. Springer, New York, pp 49–106
9. Dan A, Sitaram D, Shahabuddin P (1994) Scheduling Policies for an On-Demand Video Server with Batching. In: Proceedings of ACM Multimedia, October 1994, San Francisco, Calif. ACM Press, New York, NY, pp 15–23
10. Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York
11. Garofalakis MN, Ioannidis YE (1996) Multidimensional Resource Scheduling for Parallel Queries. In: Jagadish HV, Mumick IS (eds) Proceedings of the ACM SIGMOD International Conference on Man-

agement of Data, June 1996, Montreal, Canada. ACM Press, New York, NY, pp 365–376

12. Garofalakis MN, Özden B, Silberschatz A (1997) Resource Scheduling in Enhanced Pay-Per-View Continuous Media Databases. In: Jarke M et al. (eds) Proceedings of the 23rd International Conference on Very Large Data Bases, August 1997, Athens, Greece. Morgan Kaufmann, San Francisco, CA, pp 516–525
13. Gemmell JD, Vin HM, Kandlur DD, Rangan PV, Rowe LA (1995) Multimedia Storage Servers: A Tutorial. IEEE Comput 28(5):40–49
14. Golubchik L, Lui JCS, Muntz RR (1996) Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-on-Demand Storage Servers. Multimedia Syst 4(3):140–155
15. Han C-C, Hou C-J, Lin K-J (1996) DistanceConstrained Scheduling and Its Applications to Real-Time Systems. IEEE Trans Comput 45(7):814–826
16. Ibarra OH, Kim CE (1975) Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. J ACM 22(4):463–468
17. Kamath M, Ramamritham K, Towsley D (1995) Continuous Media Sharing in Multimedia Database Systems. In: Ling TW, Masunaga Y (eds) Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA), April 1995, Singapore. World Scientific Press, Singapore, pp 79–86
18. Knuth DE (1981) The Art of Computer Programming, Vol. 2 (Seminumerical Algorithms). Addison-Wesley, Reading, Mass.
19. Lawler EL (1979) Fast Approximation Algorithms for Knapsack Problems. Math Oper Res 4(4):339–356
20. Leung MYY, Lui JCS, Golubchik L (1997) Buffer and I/O Resource Pre-allocation for Implementing Batching and Buffering Techniques for Video-on-Demand Systems. In: Gray A, Larson P (eds) Proceedings of the Thirteenth International Conference on Data Engineering, April 1997, Birmingham, U.K. IEEE-CS Press, Los Alamitos, CA, pp 344–353
21. Little TDC, Venkatesh D (1995) Popularity-Based Assignment of Movies to Storage Devices in a Video-on-Demand System. Multimedia Sys 2:280–287
22. Liu CL, Layland JW (1973) Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J ACM 20(1): 46–61
23. Martin C, Narayanan PS, Özden B, Rastogi R, Silberschatz A (1996) The Fellini Multimedia Storage Server. In: Chung SM (ed) Multimedia Information Storage and Management. Kluwer Academic, Dordrecht, pp 117–146
24. Özden B, Biliris A, Rastogi R, Silberschatz A (1994) A Low-Cost Storage Server for Movie-on-Demand Databases. In: Bocca J et al. (eds) Proceedings of the 20th International Conference on Very Large Data Bases, September 1994, Santiago, Chile. Morgan Kaufmann, San Francisco, CA, pp 594–605
25. Özden B, Biliris A, Rastogi R, Silberschatz A (1995a) A Disk-Based Storage Architecture for Movie-on-Demand Servers. Inf Syst 20(6):465–482
26. Özden B, Rastogi R, Silberschatz A (1995b) A Framework for the Storage and Retrieval of Continuous Media Data. In: Proceedings of the International Conference on Multimedia Computing and Systems, May 1995, Washington, D.C. IEEE-CS Press, Los Alamitos, CA, pp 2–13
27. Özden B, Rastogi R, Silberschatz A, Martin C (1995c) Demand Paging for Movie-on-Demand Servers. In: Proceedings of the International Conference on Multimedia Computing and Systems, May 1995, Washington, D.C. IEEE-CS Press, Los Alamitos, CA, pp 264–272
28. Özden B, Rastogi R, Silberschatz A (1995d) Disk Striping in Video Server Environments. IEEE Data Eng Bull (Special Issue on Multimedia Information Systems) 18(4):4–16
29. Özden B, Rastogi R, Silberschatz A (1996a) Disk Striping in Video Server Environments. In: Proceedings of the International Conference on Multimedia Computing and Systems, June 1996, Hiroshima, Japan. IEEE-CS Press, Los Alamitos, CA, pp 580–589
30. Özden B, Rastogi R, Silberschatz A (1996b) On the Design of a Low-Cost Video-on-Demand Storage System. Multimedia Syst 4(1):40–54
31. Özden B, Rastogi R, Silberschatz A (1996b) The Storage and Retrieval of Continuous Media Data. In: Subrahmanian VS, Jajodia S (eds) Multimedia Database Systems: Issues and Research Directions", Springer, Berlin Heidelberg New York, pp 237–261
32. Özden B, Rastogi R, Silberschatz A (1997) Periodic Retrieval of Videos from Disk Arrays. In: Gray A, Larson P (eds) Proceedings of the Thirteenth International Conference on Data Engineering, April 1997, Birmingham, U.K. IEEE-CS Press, Los Alamitos, CA, pp 333–343
33. Patterson DA, Gibson GA, Katz RH (1988) A Case for Redundant Arrays of Inexpensive Disks (RAID). In: Clifford J et al. (eds) Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1988, Chicago, Ill. ACM Press, New York, NY, pp 109–116
34. PRECEPT Software, Inc (1998) IP/TV Datasheets. PRECEPT Software, Inc. (`http://www.precept.com/datasheets/html/iptvds1.htm`)
35. Rangan PV, Vin HM (1991) Designing File Systems for Digital Video and Audio. In: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, October 1991, Monterey, Calif. ACM Press, New York pp 81–94
36. Rangan PV, Vin HM (1993) Efficient Storage Techniques for Digital Continuous Multimedia. IEEE Trans Knowl Data Eng 5(4):564–573
37. Sahni S (1975) Approximate Algorithms for the 0/1 Knapsack Problem. J ACM 22(1):115–124
38. Shachnai H, Yu PS (1995) The Role of Wait Tolerance in Effective Batching: A Paradigm for Multimedia Scheduling Schemes. Technical Report RC 20038 (88607). IBM Research Division, Yorktown Heights, NY
39. Shi W, Ghandeharizadeh S (1997) Buffer Sharing in Video-On-Demand Servers. ACM SIGMETRICS Bull 25(2):13–20
40. Silberschatz A, Galvin P (1994) Operating System Concepts. Addison-Wesley, Reading, Mass.
41. Stankovic JA, Ramamritham K (1993) Advances in Real-Time Systems. IEEE CS Press, Los Alamitos, Calif.
42. Wei WD, Liu CL (1983) On a Periodic Maintenance Problem. Oper Res Lett 2(2):90–93
43. Yu C, Sun W, Bitton D, Yang Q, Bruno R, Tullis J (1989) Efficient Placement of Audio Data on Optical Disks for Real-Time Applications. Commun ACM 32(7):862–871

Appendix

Proofs of theoretical results

Proof of Lemma 3.1. First observe that, by the specific form of clip values and sizes, the inequality $p_i \geq p_j$ is equivalent to $\mathsf{value}(C_i) \geq \mathsf{value}(C_j)$, and it is also equivalent to $\mathsf{size}(C_i) \geq \mathsf{size}(C_j)$.

Consider the heuristic H_1 that always selects the *first* n_{disk} bins from the first-fit packing described in PACKCLIPS. Obviously, PACKCLIPS will always do at least as good as H_1; that is $V_H \geq V_{H_1}$. Consider the clips in order of decreasing value density (which is also decreasing size) and let C_m be the first clip placed in the n_{disk} + 1th bin by H_1. At that point in time, the following observations can be made.

1. Each of the first n_{disk} bins is *at least half-full* (by virtue of first-fit and decreasing clip siześ).
2. The bin capacity filled up to now by H_1 was filled at a value density that is greater than or equal to the density that the optimal algorithm would use to fill the corresponding part of the bin(s) (by the order of clip placement).

Let $V_{H_1}(m)$ denote the total value of the first $m - 1$ clips in the list (already packed in the first n_{disk} bins). By observations (1) and (2) it is clear that the remaining capacity at that point cannot contribute more than $V_{H_1}(m)$ to the optimal solution. From this and observation (2) we conclude that

$$V_{OPT} \leq V_{H_1}(m) + V_{H_1}(m) \leq 2 \cdot V_{H_1} \leq 2 \cdot V_H \,.$$

Proof of Lemma 3.2. We extend the length function $l()$ (defined in Sect. 3.2) to sets of d-dimensional vectors S as follows: Let $\mathbf{w} = \sum_{\mathbf{v} \in S} \mathbf{v}$, then $l(S) = \max_{1 \leq i \leq d}\{\mathbf{w}[i]\}$ (i.e., the maximum component of the vector sum of all elements of S). The following result establishes an important property of the vector length function that is used in our worst case analysis. Its proof is based on a "pigeonhole argument" (see, for example, Garofalakis and Ioannidis 1996).

Proposition A.1. Let S be a collection of d-dimensional vectors and let $S_1, \ldots, S_n$ be *any* partition of S. Then,

$$\frac{\sum_{i=1}^{n} l(S_i)}{d} \leq l(S) \leq \sum_{i=1}^{n} l(S_i)\,.$$

Consider the items in order of decreasing value density and let C_m be the first item that is placed in the n_{disk}+1th bin by PackClips. Let B_j, S_j denote the fraction of bandwidth and storage capacity (respectively) of the j-th disk ($1 \leq j \leq n_{disk}$) that is used by clips mapped onto that disk. We also use $\mathbf{d}_j$ to describe the vector with components B_j and S_j.

The first-fit rule used by our heuristic ensures that when C_m is pushed to the extra disk we have

$$\max\{B_j + \mathbf{s}_m[1], S_j + \mathbf{s}_m[2]\} > 1$$

for every disk $j = 1, \ldots, n_{disk}$.

Since decreasing value density implies decreasing values for $\mathbf{s}_i[1]$ and $\mathbf{s}_i[2] \leq \frac{1}{2}$, the above condition implies that

$$B_j > 1 - \frac{1}{2} \qquad \text{or} \qquad S_j > 1 - \frac{1}{2}$$

or, equivalently: $l(\mathbf{d}_j) > \frac{1}{2}$ for every disk $j = 1, \ldots, n_{disk}$.

Let S_H denote the sets of clips packed in the first n_{disk} bins by our heuristic. Let S_{OPT} be the set of clips scheduled by an optimal policy. Consider the partitions of these two sets $S_H^1, \ldots, S_H^{n_{disk}}$ and $S_{OPT}^1, \ldots, S_{OPT}^{n_{disk}}$ produced by our heuristic and the optimal policy, respectively. We also use these symbols to represent the corresponding collection of size vectors (i.e., S_H^j is also the collection of size vectors packed in the j-th disk). Using Proposition A.1 and the analysis outlined in the previous paragraph, it is easy to see that

$$n_{disk} \geq \sum_{j=1}^{n_{disk}} l(S_{OPT}^j) \geq \frac{1}{2} \cdot \sum_{C_i \in S_{OPT}} l(\mathbf{s}_i), \qquad \text{and}$$

$$\frac{n_{disk}}{2} < \sum_{j=1}^{n_{disk}} l(S_H^j) \leq \sum_{C_i \in S_H} l(\mathbf{s}_i)\,,$$

which implies

$$\sum_{C_i \in S_H} l(\mathbf{s}_i) > \frac{1}{4} \cdot \sum_{C_i \in S_{OPT}} l(\mathbf{s}_i).$$

Thus, if we consider the optimal solution as a "bin" with total capacity of $\sum_{C_i \in S_{OPT}} l(\mathbf{s}_i)$, the above inequality guarantees that the PackClips heuristic will fill up more than 1/4-th of that capacity (before placing a clip in disk n_{disk}+1). Further, by the order of clip placement, we know that that 1/4-th was filled at the *maximum possible density* for the given set of clips. Thus, the total "capacity" of the optimal solution cannot possibly contribute more than $4 \cdot value(S_H)$ to the total value. This obviously implies the result.

Proof of Lemma 5.1. Assume that the retrieval of C_i from a particular disk is initiated at time slot t_i. Then, the slots (partially) occupied by the retrieval of C_i from that disk are given by the equation $t = t_i + k \cdot n_i + j \cdot n_{disk}$, where $k \geq 0$ and $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$.

Thus, given t_i, the slots for clip C_i are characterized by a set of ≥ 1 moduli: $u_{ij} = t \bmod n_i = (t_i + j n_{disk}) \bmod n_i$ for $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$. The Chinese Remainder Theorem can now be directly used to obtain the result.

Proof of Lemma 5.2. Our proof uses the following ancillary result, which is a direct consequence of the *Generalized Chinese Remainder Theorem* (Knuth 1981). It provides necessary and sufficient conditions under which collisions will occur in a fixed schedule of periodic clip retrieval tasks.

Lemma A.1. Consider a specific disk and let t_i denote the start time for the retrieval of C_i and $n = \text{lcm}(n_1, \ldots, n_N)$ (i.e., the least common multiple of n_1, ..., n_N). Also, let $u_{ij} = (t_i + j \cdot n_{disk}) \bmod n_i$ for $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$. Then, in any interval of length n (after all retrievals are initiated) there exists exactly one time slot where all retrievals collide *if and only if* for all $1 \leq i < k \leq N$, $u_{ij} \equiv u_{kl} \pmod{\gcd(n_i, n_k)}$ for *some* combination of $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$ and $0 \leq l < \left\lceil \frac{c_k}{n_{disk}} \right\rceil$.

Observe that, by its definition, α_i is exactly the number of *distinct* moduli ($\bmod \gcd(n_1, n_2)$) that are "occupied" by the retrieval of clip C_i. To see this, assume that the retrieval of C_i is initiated at some time slot x. Then the pattern of the time slots occupied by C_i ($\bmod \gcd(n_1, n_2)$) will start repeating itself as soon as the condition $x + k \cdot n_{disk} \equiv x \pmod{\gcd(n_1, n_2)}$ is satisfied, or equivalently $k \cdot n_{disk} = \text{multiple}(\gcd(n_1, n_2))$. This equality will be satisfied for the first time when the right-hand side equals $\text{lcm}(n_{disk}, \gcd(n_1, n_2))$, i.e., the least common multiple of n_{disk} and $gcd(n_1, n_2)$. Thus, the number of distinct moduli ($\bmod \gcd(n_1, n_2)$) occupied by C_i is either $k = \frac{\text{lcm}(n_{disk}, \gcd(n_1, n_2))}{n_{disk}} = \frac{\gcd(n_i, n_j)}{\gcd(n_i, n_j, n_{disk})}$ (using the identity $\text{lcm}(x, y) = \frac{x \cdot y}{\gcd(x, y)}$) or $\left\lceil \frac{n_i}{n_{disk}} \right\rceil$, whichever is smaller. Also observe that by the form of the retrieval patterns, these α_i moduli occupied by C_i are regularly spaced (n_{disk}($\bmod$ $\gcd(n_1, n_2)$) time slots apart).

Thus, it is easy to see that, if the condition $\alpha_1 + \alpha_2 \leq \gcd(n_1, n_2)$ is satisfied, then it is always possible to "shift" one of the patterns so that no collisions occur. On the other hand, if $\alpha_1 + \alpha_2 > \gcd(n_1, n_2)$ then a simple "pigeonhole argument" shows that the moduli equality in Lemma A.1 will always be satisfied (for some combination of l and j). Hence, the retrievals of C_1 and C_2 will collide. This completes the proof.

Proof of Lemma 5.3. The proof proceeds in a manner exactly similar to the proof of Lemma 5.2 (using the observation that $\gcd(n_i, n_j) = k$ for all $i \neq j$).

Correctness of algorithm BUILDEQUIDTREE. The correctness of BUILDEQUIDTREE relies on the following claim. Given a set of tasks, there is a collision-free schedule, if the tasks can be partitioned into n_{disk} partitions such that the following conditions hold.

1. For each partition, there is an integer d, which is a multiple of n_{disk}, that divides the periods of all the tasks in that partition.
2. Each partition can be further partitioned into subpartitions such that the following hold.
   2.1 The first subtasks of any pair of tasks C_i and C_j in each subpartition can be scheduled collision-free in time slots u_i and u_j such that $u_i \bmod \frac{d}{n_{disk}} = u_j \bmod \frac{d}{n_{disk}}$ holds.
   2.2 $\sum_{j=0}^{\frac{d}{n_{disk}}-1} s^j_{max} \leq \frac{d}{n_{disk}}$, where s^j_{max} is the maximum of the number of subtasks of any task in the subpartition.

We will now prove that, if tasks can be partitioned such that if Conditions 1 and 2 hold, one can generate a collision-free schedule by modifying each u_i in each partition j, $0 \leq j < n_{disk}$, by $u'_i = j + u_i \cdot n_{disk}$. Obviously, the subtasks of tasks within subpartitions of different partitions cannot collide. This is because for any pair of subtasks C_i and C_j in a partition, $u'_i \bmod n_{disk} = u'_j \bmod n_{disk}$ holds, where u'_i and u'_j are the time slots in which subtasks C_i and C_j are scheduled, respectively. Since the distance between two consecutive subtasks of a task is n_{disk}, all the subtasks of a task will be in the same partition (i.e., $u'_i \bmod n_{disk} = (u'_i + j \cdot n_{disk}) \bmod n_{disk}$, $0 \leq j < s_i$). Due to Condition 2.1, the first subtasks within the same subpartition do not collide. Furthermore, Condition 2.1 implies that for any pair of tasks C_i and C_j in a subpartition, $\frac{u'_i \bmod d - u'_i \bmod n_{disk}}{n_{disk}} = \frac{u'_j \bmod d - u'_j \bmod n_{disk}}{n_{disk}}$ holds. Let us enumerate the subpartitions of a partition by the value of $\frac{u'_i \bmod d - u'_i \bmod n_{disk}}{n_{disk}}$, where u'_i the time slot in which the first subtask of a task C_i in that subpartition is scheduled. Since $\frac{u'_i \bmod d - u'_i \bmod n_{disk}}{n_{disk}}$ and $\frac{(u'_i + n_{disk}) \bmod d - (u'_i + n_{disk}) \bmod n_{disk}}{n_{disk}}$ differ only by one, in such a schedule, the subtasks of a task will be in consecutive subpartitions of a partition. Once the first subtask of a task C_i is scheduled in a subpartition, due to Condition 2.2, we can ensure that in the next $s_i - 1$ subpartitions none of the tasks' first subtask is scheduled, we ensure that none of the subtasks of the tasks scheduled within a subpartition will collide with any of the subtasks scheduled within a different subpartition of the same partition. Furthermore, it is straightforward to prove that, if the first subtasks of two tasks are scheduled within the same subpartition j without collision, none of their subtasks will collide provided that none of the tasks' first subtask is scheduled in the next $s^j_{max} - 1$ subpartitions.