Resource Scheduling for Composite Multimedia Objects

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

Yannis E. Ioannidis†‡
University of Athens
yannis@di.uoa.gr

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

Abstract

Scheduling algorithms for composite multimedia presentations need to ensure that the user-defined synchronization constraints for the various presentation components are met. This requirement gives rise to task models that are significantly more complex than the models employed in scheduling theory and practice. In this paper, we formulate the resource scheduling problems for composite multimedia objects and develop novel efficient scheduling algorithms drawing on a number of techniques from pattern matching and multiprocessor scheduling. Our formulation is based on a novel *sequence packing problem*, where the goal is to superimpose numeric sequences (representing the objects' resource needs as a function of time) within a fixed capacity bin (representing the server's resource capacity). Given the intractability of the problem, we propose heuristic solutions using a two-step approach. First, we present a "basic step" method for packing two composite object sequences into a single, combined sequence. Second, we show how this basic step can be employed within different scheduling algorithms to obtain a playout schedule for multiple objects. More specifically, we present an algorithm based on Graham's list-scheduling method that is provably near-optimal for *monotonic* object sequences. We also suggest a number of optimizations on the base list-scheduling scheme. Preliminary experimental results confirm the effectiveness of our approach.

1 Introduction

Next generation database systems will need to provide support for various forms of multimedia data such as images, video, and audio. These new data types differ from conventional alphanumeric data in their characteristics, and hence require different techniques for their organization and management. A fundamental issue is that digital video and audio *streams* consist of a sequence of media quanta (video frames or audio samples) which convey meaning only when presented continuously in time. Hence, a multimedia database server needs to provide a guaranteed level of service for accessing such *continuous media* (CM) streams in order to satisfy their pre-specified real-time delivery rates and ensure *intra-media continuity*. Given the limited amount of resources (e.g., memory and disk bandwidth), it is a challenging problem to design effective resource scheduling algorithms that can provide on-demand support for a large number of concurrent continuous media clients.

An important requirement for multimedia database systems is the ability to dynamically compose new multimedia objects from an existing repository of CM streams. Temporal and spatial primitives specifying the relative timing and output layout of component CM streams provide perhaps the most powerful and natural method of authoring such *composite* multimedia presentations. Thus, to compose tailored multimedia presentations, a user might define temporal dependencies among multiple CM streams having various length and display bandwidth requirements. For example, a story for the evening news can start out by displaying a high resolution video clip with concurrent background music and narration added after an initial delay. After some time into the story, the video screen is split and a new video clip starts playing on the left half of the screen. After the second video clip ends, the narration stops and the story comes to a conclusion with the display of the first clip and the background music.

In the presence of such composite multimedia objects, a scheduling algorithm must ensure that the *inter-media synchronization* constraints defined by the temporal relationships among CM components are met. Handling these synchronization constraints requires a task model that is significantly more complex than the models employed in scheduling theory and practice [13, 14, 30]. More specifically, composite multimedia objects essentially correspond to *resource-constrained tasks with time-varying resource demands*. Resource constraints come from the limited amount of server resources available to satisfy the requirements of CM streams and time-variability stems from the user-defined inter-media synchronization requirements. This is a task model that has not been previously studied in the context of deterministic scheduling theory. Further-

* Part of this work was performed while the first author was visiting Bell Laboratories.

† On leave from the University of Wisconsin–Madison.

‡ Partially supported by the National Science Foundation under Grant IRI-9157368 (PYI Award), and by grants from IBM, DEC, HP, AT&T, Informix, and Oracle.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

more, despite the obvious importance of the problem for multimedia database systems, our work appears to be the first systematic study of the problems involved in scheduling multiple composite multimedia objects. We suspect that this is due to the difficulty of the problems, most of which are non-trivial generalizations of $\mathcal{NP}$-hard optimization problems. Finally, note that although our discussion in this paper is primarily geared towards composite objects, our task model also exactly captures the problem of scheduling the retrievals *variable bit rate* streams, that is, CM streams whose bandwidth requirements can vary over time. This is also a very important application of our scheduling framework, since real-life CM data is nearly always variable rate.

To the best of our knowledge, none of today's multimedia storage servers offer any clever scheduling support for composite multimedia presentations. The approach typically employed is to reserve server resources based on the *maximum (i.e., worst-case)* resource demand over the duration of a composite presentation. Examples of systems using this worst-case resource reservation method include the $\mathcal{F}$*ellini* and CineBlitz multimedia storage servers developed at Bell Labs [22], Starlight's StarWorks (`http://www.starlight.com/`), and Oracle's Media Server (`http://www.oracle.com/`). Conceptually, this approach is equivalent to identifying the resource requirements of the presentation over time with their enclosing *Minimum Bounding Rectangle (MBR)*. Although this simplification significantly reduces the complexity of the relevant scheduling problems, it suffers from two major deficiencies.

- The *volume* (i.e., resource-time product [16]) in the enclosing MBR can be significantly larger than the actual requirements of the composite object. This can result in wasting large fractions of precious server resources, especially for relatively "sparse" composite objects.
- The MBR simplification "hides" the timing structure of individual streams from the scheduler, making it impossible to improve the performance of a schedule through clever use of memory buffers.

The only work to address some of the research issues in scheduling composite multimedia objects is that of Chaudhuri et al [7] and Shahabi et al. [29]. However, their work has focused on (1) the use of memory to resolve the problem of "internal contention", which occurs when the temporal synchronization constraints cause stream retrievals for a single object to collide; and, (2) the development of heuristic memory management policies to distribute a fixed amount of server memory among multiple competing objects. More specifically, Chaudhuri et al. suggest a conservative and a greedy heuristic for allocating memory among multiple *binary* composite objects, under the assumption of regular, round-robin striping of component streams [7]. However, extending their heuristics to general, n-ary objects appears to be problematic [7]. Shahabi et al. show how these conservative and greedy methods can be adapted to the problem of resolving internal contention in a single n-ary composite object, again assuming round-robin layout [29]. Although the authors outline some ideas on how to actually schedule multiple composite objects, they offer no concrete algorithmic solutions for the problem. Furthermore, their development is based on the assumption of a round-robin distribution of stream fragments across disks, whereas we assume a more abstract "black-box" model of server disk bandwidth.

In this paper, we formulate the resource scheduling problems for composite multimedia objects and we develop novel efficient scheduling algorithms, drawing on a number of techniques from pattern matching and multiprocessor scheduling. Our formulation is based on a novel *sequence packing problem*, where the goal is to superimpose numeric sequences (representing the objects' resource needs as a function of time) within a fixed capacity bin (representing the server's resource capacity). We propose heuristic algorithms for the sequence packing problem using a two-step approach. First, we present a "basic step" method for packing two object sequences into a single, combined sequence. Second, we show how this basic step can be employed within different scheduling heuristics to obtain a playout schedule for multiple composite objects. More specifically, we examine greedy scheduling heuristics based on the general list-scheduling ($\mathcal{LS}$) methodology of Graham [18, 13]. We show that although $\mathcal{LS}$ schemes are *provably near-optimal* for packing *monotonic* sequences, they can have poor worst-case performance when the monotonicity assumption is violated. Based on this result, we: (1) suggest methods for improving the behavior of simple $\mathcal{LS}$ through the use of extra memory buffers; and, (2) propose a novel family of more clever scheduling algorithms, termed *list-scheduling with backtracking* ($\mathcal{LSB}$), that try to improve upon simple $\mathcal{LS}$ by occasional local improvements to the schedule. Preliminary experimental results with randomly generated composite objects show that our $\mathcal{LS}$ strategy offers excellent average-case performance compared to both an MBR-based approach and the optimal solution. Finally, we briefly discuss our ongoing work on how the idea of *stream sharing* (i.e., allowing several presentations to share component streams) can be exploited to improve the quality of a schedule.

2 Definitions and Problem Formulation

2.1 Composite Objects and Object Sequences

A composite multimedia object consists of multiple CM streams tied together through spatial and temporal primitives. Since the spatial layout of the output is predetermined by the user and does not affect the resource bandwidth requirements of CM streams, we concentrate on the temporal aspects of CM composition. Following the bulk of the multimedia systems literature, we also concentrate on the server disk bandwidth resource which is typically the bottleneck for multimedia applications [7, 25, 29]. To simplify the presentation, we assume that the stream resource demands have been normalized to $[0, 1]$ using the aggregate disk bandwidth of the server B. We also assume that the time scale is *discrete* so that both the lengths of

CM streams and their lag parameters have integer values. This is usually the case in practice, since most multimedia storage servers employ a *round-based* data retrieval scheme and thus timing can only be specified at the granularity of a round's length, which is typically very small (a few seconds) [25]. Of course, finer-grain synchronization can always be implemented using extra memory buffering [29].

Following Chaudhuri et al. [7], we define an n_i-ary composite multimedia object C_i as a $(2n_i - 1)$-tuple $< X_{i_1}, X_{i_2}, \ldots, X_{i_{n_i}}, t_{i_2}, \ldots, t_{i_{n_i}} >$ where the X_{i_j}'s denote the component CM streams (in order of increasing start times) and t_{i_j} denotes the *lag* factor of X_{i_j} with respect to the beginning of the display of X_{i_1} (i.e., the beginning of the composite object). This definition covers the 13 qualitative temporal interval relationships of Allen [1] and also allows us to specify quantitative temporal constraints. Figure 1(a) depicts a 4-ary object corresponding to the news story example mentioned in Section 1, consisting of two overlapping video clips with background music and narration. The height of each stream in Figure 1(a) corresponds to its bandwidth requirement, and the length corresponds to its duration (in general, the x-axis represents time).

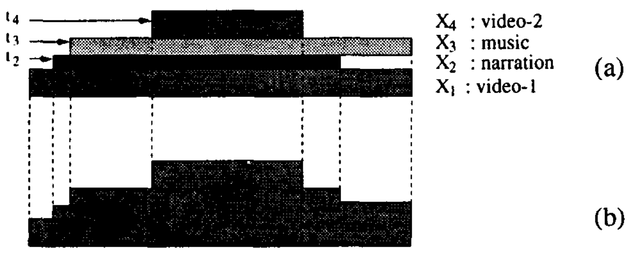


Figure 1: (a) A 4-ary composite multimedia object. (b) The corresponding object sequence.

For each component CM stream X_{i_j} of C_i, we let $l(X_{i_j})$ denote the time duration of the stream and $r(X_{i_j})$ denote its resource bandwidth requirements. Similarly, we let $l(C_i)$ denote the duration of the entire composite object C_i, i.e., $l(C_i) = \max_j\{t_{i_j} + l(X_{i_j})\}$, and $r(C_i, t)$ denote the bandwidth requirements of C_i at the t^{th} time slot after its start $(0 \leq t < l(C_i))$. Table 1 summarizes the notation used throughout the paper with a brief description of its semantics. Detailed definitions of some of these parameters are given in the text. Additional notation will be introduced when necessary.

The bandwidth requirements of our example news story object can be represented as the *composite object sequence* depicted graphically in Figure 1(b), where each element of the sequence corresponds to the object's bandwidth demand at that point in time (i.e., during that time unit). Note that the rising and falling edges in a composite object sequence correspond to CM streams starting and ending, respectively. Essentially, the object sequence represents $r(C_i, t)$, that is, the (varying) bandwidth requirements of the object as a function of time t. Since our scheduling problem focuses on satisfying the bandwidth requirements of objects, we will treat the terms "composite object" and

Table 1: Notation

Parameter	Semantics
B	Aggregate server disk bandwidth (in bits per sec – bps)
T	Length of a time unit (i.e., round) (in sec)
C_i	Composite multimedia object
n_i	Number of component streams in C_i
$l(C_i)$	Length (i.e., time duration) of C_i
$r(C_i, t)$	Bandwidth requirement of C_i at time slot t $(0 \leq t < l(C_i))$
$r_{max}(C_i)$	Maximum bandwidth requirement of C_i
X_{i_j}	The j^{th} component stream of C_i
$l(X_{i_j})$, $r(X_{i_j})$	Length and bandwidth requirement of stream X_{i_j}
k_i	Number of constant bandwidth "blocks" in the run-length compressed form of C_i
(l_{i_j}, r_{i_j})	Length and bandwidth requirement of the j^{th} run-length "block" of C_i
$V(C_i)$	Volume (i.e., resource-time product) of C_i
$d(C_i)$	Density of C_i

"sequence" as synonymous in the remainder of the paper.

Typically, CM streams tend to last for long periods of time. This means that using the full-length, $l(C_i)$-element object sequence for representing and scheduling a composite multimedia object is a bad choice for the following two reasons. First, these full-length sequences will be very long (e.g., a 2-hour presentation will typically span thousands of rounds/time units). Second, full-length object sequences will be extremely redundant and repetitive since they only contain a small number of transition points. For our purposes, a more compact representation of composite objects can be obtained by using the *run-length compressed* form of the object sequences [3]. Essentially, the idea is to partition the composite object into "blocks" of constant bandwidth requirement and represent each such block by a pair (l_{i_j}, r_{i_j}), where r_{i_j} represents the constant requirement of the object over a duration of l_{i_j} time units. This process is shown graphically in Figure 1(b). Thus, we can represent the n-ary composite object C_i in a compact manner by the sequence: $< (l_{i_1}, r_{i_1}), \ldots, (l_{i_{k_i}}, r_{i_{k_i}}) >$, where $k_i << l(C_i)$. In fact, $k_i \leq 2 \cdot n_i - 1$, where n_i is the number of component CM streams in C_i.

We define the *volume (V)* of a composite object C_i as the total resource-time product over the duration of C_i [16]. More formally, $V(C_i) = \sum_{j=1}^{k_i} l_{i_j} r_{i_j}$. The *density (d)* of a composite object C_i is defined as the ratio of the object's volume to the volume of its MBR, i.e., $d(C_i) = \frac{V(C_i)}{l(C_i) \cdot r_{max}(C_i)}$.

2.2 Using Memory to Change Object Sequences: Stream Sliding

Although inter-media synchronization constraints completely specify the relative timing of streams at presentation time, the scheduler can use extra memory buffers[1] to alter an object's retrieval sequence. The idea is to use additional memory to buffer parts of streams that have been retrieved before they are actually needed in the presentation and play them out *from memory* at the appropriate time. This method, termed ***stream sliding***, was originally introduced by Chaudhuri et al. for resolving internal contention under the assumption of round-robin striping [7]. The general method is depicted in Figure 2(a) which shows our example 4-ary news story object with the second video clip (stream X_4) ***upslided*** by x time units. In this example, the server needs to use an extra $x \cdot T \cdot B \cdot r(X_4)$ bits of memory in order to support the object playout as required by the user (Figure 1(a)). Since it is possible for the amount of upsliding to exceed the actual length of the stream (i.e., $x > l(X_4)$), the general expression for the amount of memory required to upslide X_4 by x is $\min\{x, l(X_4)\} \cdot T \cdot B \cdot r(X_4)$. This expression says that if $x > l(X_4)$, then we only need enough memory to buffer the entire stream X_4. (Note that multiplying by the server bandwidth B is necessary, since $r()$ is normalized using B.)

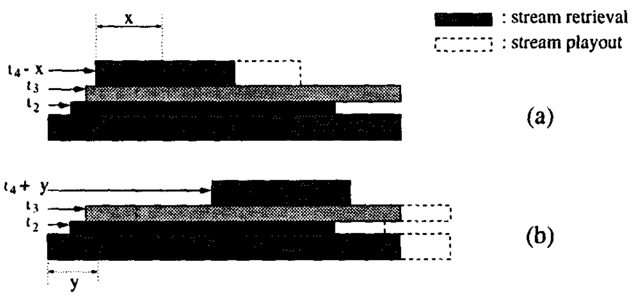


Figure 2: Upsliding (a) and downsliding (b) stream X_4.

Similarly, the server may also choose to *downslide* a stream, which means that the retrieval of the stream starts *after* its designated playout time in the presentation (Figure 2(b)). Downsliding introduces latency in the composite object presentation, since it is clearly impossible to start the playout of a stream before starting its retrieval. Thus, in Figure 2(b), the entire presentation must be delayed by y time units. This also means that once a stream is downslided, *all the other streams* must be buffered unless they are also downslided by the same amount. Because of these two problems upsliding is preferable to downsliding, whenever both options are available [7, 29].

2.3 Our Scheduling Problem: Sequence Packing

Given a collection of composite objects to be scheduled using the server's resources, a *schedule* is an assignment of start times to these objects so that, at any point in time, the bandwidth and memory requirements of concurrent presentations (to, perhaps, different users) do not violate the resource capacities of the server. In this paper, we concentrate on the problem of *off-line makespan (i.e., response time) minimization*, in which the objective is to minimize the overall schedule length for a given collection of objects (i.e., tasks) [13, 18]. Prior scheduling theory research has shown how to employ solutions to this problem for both *on-line* response time minimization (where tasks arrive dynamically over time) and on-line average response time minimization [6, 19, 30].

Ignoring the flexibilities allowed in stream synchronization through additional memory buffering (i.e., sliding), the bandwidth requirements of each object to be scheduled are completely specified by its resource demand sequence (Figure 1(b)). Thus, assuming sliding is not an option, our scheduling problem can be abstractly defined as follows.

- **Given:** A collection of (normalized) composite object sequences $\{C_i\}$ over $[0, 1]$.
- **Find:** A start time slot $s(C_i)$ for each i, such that for each time slot t

$$\sum_{\{C_i : s(C_i) \le t < s(C_i) + l(C_i)\}} r(C_i, t - s(C_i)) \le 1,$$

and $\max_i\{s(C_i) + l(C_i)\}$ is minimized.

Conceptually, this corresponds to a *sequence packing* problem, a non-trivial generalization of traditional $\mathcal{NP}$-hard optimization problems like bin packing and multiprocessor scheduling that, to the best of our knowledge, has not been previously studied in the combinatorial optimization literature [9, 10, 18, 13]. In bin packing terminology, we are given a set of items (normalized object sequences) that we want to pack within unit-capacity bins (server bandwidth) so that the total number of bins (makespan, used time slots) is minimized. Our sequence packing problem also generalizes *multi-dimensional* bin packing models known to be intractable, like *orthogonal rectangle packing* [4, 5, 11] (a rectangle is a trivial, constant sequence) and *vector packing* [14, 21] (vectors are fixed length sequences with start times restricted to bin boundaries). Note that rectangle packing algorithms are directly applicable when the MBR simplification is adopted. However, it is clear that this simplification can result in wasting large fractions of server bandwidth when the object densities are low. Given the inadequacy of the MBR simplification and the intractability of the general sequence packing formulation, we propose novel efficient heuristics for scheduling composite object sequences using a combination of techniques from pattern matching and multiprocessor scheduling.

Sliding further complicates things, since it implies that the scheduler is able to *modify* the composite object sequences at the cost of extra memory. Given a set of object sequences and a finite amount of memory available at the server, the problem is how to utilize memory resources for sliding various object streams around (i.e., modifying the object sequences) so that the scheduler can effectively minimize some scheduling performance metric such as schedule length or average response time. This is obviously a very complex problem that, in many ways, generalizes recently proposed *malleable* multiprocessor scheduling problems [32]. The general sliding problem, as outlined above,

[1] Assuming round-based retrieval of streams with a round length of T, each stream X_{i_j} requires a minimum buffering of $2 \cdot T \cdot B \cdot r(X_{i_j})$ during its retrieval [25].

has yet to be addressed in the scheduling or multimedia literature.

Our results for the sequence packing problem indicate that simple, greedy scheduling algorithms based on Graham's list-scheduling method [18] can guarantee *provably* near-optimal solutions, as long as the sequences are monotonic. On the other hand, we show that list-scheduling can perform poorly when the monotonicity assumption is violated. Based on this result, we examine the problem of exploiting extra memory and sliding to make object sequences monotonic. Although this problem is itself $\mathcal{NP}$-hard, we propose a polynomial-time approximate solution.

3 Algorithms for Sequence Packing

Our approach is based on the observation that the result of packing a subset of the given object sequences is itself an object sequence. Thus, we begin by presenting a "basic step" algorithm for obtaining a valid packing of two sequences. We then show how this method can be employed within two different scheduling heuristics to obtain a playout schedule for multiple composite objects.

3.1 The Basic Step: Packing Two Sequences

Our basic algorithmic step problem can be abstractly described as follows. We are given two (normalized) object sequences C_o (a new object to be scheduled) and C_p (the partial schedule constructed so far) over $[0, 1]$. We want to determine a valid packing of the two sequences, that is, a way to superimpose C_o over C_p that respects the unit capacity constraint (i.e., all elements of the combined sequence are less than or equal to 1). Given that our overall scheduling objective is to minimize the length of the resulting composite sequence, the presentation of this section assumes a "greedy" basic step that searches for the *first point* of C_p at which C_o can be started without causing capacity constraints to be violated. Since, as we argued in Section 2.1, the full-length representation of the object sequences is very inefficient we assume that both object sequences are given in their run-length compressed form. That is, $C_o = < (l_{o_1}, r_{o_1}), \ldots, (l_{o_{k_o}}, r_{o_{k_o}}) >$ and $C_p = < (l_{p_1}, r_{p_1}), \ldots, (l_{p_{k_p}}, r_{p_{k_p}}) >$. In Figure 3, we present an algorithm, termed FINDMIN, for performing the basic sequence packing step outlined above. FINDMIN is essentially a "brute-force" algorithm that runs in time $O(k_o \cdot k_p)$, where k_o, k_p are the lengths of the compressed object sequences.

Algorithm FINDMIN(C_o, C_p)

```
Input: Sequences $C_o$, $C_p$ over $[0,1]$ in run-length compressed
    form (i.e., $C_o = < (l_{o_1}, r_{o_1}), \ldots, (l_{o_{k_o}}, r_{o_{k_o}}) >$ and
    $C_p = < (l_{p_1}, r_{p_1}), \ldots, (l_{p_{k_p}}, r_{p_{k_p}}) >$).
Output: The least $0 \le k \le l(C_p)$ such that $C_o$ can be validly
    superimposed over $C_p$ starting at time slot $k$.
1. For each constant bandwidth block $< l_{o_j}, r_{o_j} >$ of $C_o$,
   determine the set of feasible starting points $S_{o_j}$ for $<
   l_{o_j}, r_{o_j} >$ over $C_p$. For a given $j$, this can be done in time
   $O(k_p)$ and the result is a union of $m_j = O(k_p)$ disjoint
   temporal intervals:
       $S_{o_j} = [a_{j1}, b_{j1}) \cup \ldots \cup [a_{jm_j}, b_{jm_j})$,
   where $b_{jm_j} = \infty$ (at the end of the current partial schedule
   $C_p$).
2. Let $I_1 = S_{o_1}$ and $l_{prev} = 0$. For $j = 2$ to $k_o$ do
   2.1. Set $l_{prev} = l_{prev} + l_{o_{j-1}}$ and
        $I_j = I_{j-1} \cap ( [a_{j1} - l_{prev}, b_{j1} - l_{prev}) \cup \ldots$
              $\cup [a_{jm_j} - l_{prev}, b_{jm_j} - l_{prev}) )$.
   Let $|S|$ denote the number of intervals in $S$. Since the inter-
   vals in the unions $S_{o_j}$ are disjoint and in sorted order, each
   intersection $I_j$ can be computed in time $O(|I_{j-1}| + |S_{o_j}|)$
   using a MERGE-like algorithm, and a simple argument can
   establish that $|I_j| \le \min\{|I_{j-1}|, |S_{o_j}|\} = O(k_p)$.
3. At this point, $I_{k_o}$ contains the entire set of feasible starting
   time slots for $C_o$. Return the earliest slot in $I_{k_o}$.
```

Figure 3: Algorithm FINDMIN

Since our composite obect sequences can be seen as *patterns* over the alphabet $[0, 1]$, it is natural to ask whether or not ideas from the area of *pattern matching* can be used to make our basic algorithmic step more efficient. As in most pattern matching problems [3], the requirement that all "characters" of both patterns must be examined imposes a linear, i.e., $O(k_o + k_p)$ (since we are dealing with the compressed representations), lower bound on the running time of any algorithm. The question is whether or not this lower bound is attainable by some strategy. An equivalent formulation of our basic step packing problem comes from considering the *complementary sequence* $\overline{C_p}$ of the partial schedule C_p, which informally, corresponds to the sequence of free server bandwidth over time. More formally, $\overline{C_p}$ is determined by defining $r(\overline{C_p}, t) = 1 - r(C_p, t)$, for each time slot t. Thus, our problem is equivalent to finding the earliest time slot at which C_o is completely "covered"/dominated by $\overline{C_p}$ (Figure 4(a)). This is a problem that has received some attention from the pattern matching research community. Amir and Farach define the above problem for *uncompressed* patterns as the *"smaller matching"* problem and give an algorithm that runs in time $O\left(l(C_p) \cdot \sqrt{l(C_o)} \cdot \log l(C_o)\right)$, where $l()$ denotes uncompressed sequence lengths (Table 1) [2]. Muthukrishnan and Palem show that this is in fact a lower bound on the time complexity of the problem in a special, yet powerful convolution-based model; their result implies that faster algorithms for the smaller matching problem would imply a faster method than the Fast Fourier Transform for certain generalizations of convolutions [24]. None of these papers addressed the problem when the *run-length compressed* forms of the patterns are used. Furthermore, it is not clear whether the rather complicated algorithm of Amir and Farach will outperform a straightforward $O(l(C_p) \cdot l(C_p))$ solution in a practical implementation [12]. Finally, even if the "optimal" method of Amir and Farach could be extended to run-length compressed patterns, the asymptotic performance improvement with respect to FINDMIN would only be $O(\frac{\sqrt{k_o}}{\log k_o})$, a small gain since the number of streams in a single composite object k_o is typically bounded by a small constant.

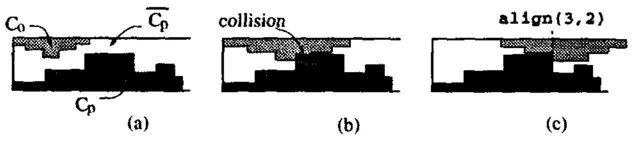


Figure 4: (a)The "smaller matching" analogy. (b) A collision with a bitonic C_o. (c) Resolving the collision by align-ing (Algorithm BITONIC-FINDMIN).

Thus, prior results from the pattern matching community assure us that our "brute-force" FINDMIN algorithm is a reasonably good strategy for the basic sequence packing step for general object sequences. However, for special cases of object sequences (C_o), we may still be able to come up with faster algorithms. We now present such an algorithm for the special case of *bitonic object sequences*. Informally, a sequence is bitonic if it can be partitioned into a monotonically increasing prefix followed by a monotonically decreasing prefix. This means that no new component streams can be initiated after the end of a stream in the presentation. Although one can argue that bitonic objects are rather common in multimedia practice (e.g., our example news story composite object shown in Figure 1 is bitonic), our method can also be used within more complex basic step algorithms for general sequences. The idea is to partition objects into a (small) number of bitonic components and schedule each component using our improved strategy for bitonic objects. We will not pursue this idea further in this paper.

The basic operation of our improved algorithm for bitonic objects is similar to that of the Knuth-Morris-Pratt string matching algorithm, in that it shifts the C_o pattern over C_p until a "fit" is discovered [20]. The crucial observation is that, for bitonic C_o patterns, we can perform this shifting in an efficient manner, without ever having to backtrack on C_p. This is done as follows. Consider a particular alignment of C_o and C_p and let (l_{p_j}, r_{p_j}) be the first block of C_p at which a "collision" (i.e., a capacity violation) occurs. Then the earliest possible positioning of C_o that should be attempted (without losing possible intermediate positions) is that which aligns the right endpoint of block (l_{p_j}, r_{p_j}) with that of block $(l_{o_{i_j}}, r_{o_{i_j}})$, where i_j is the index of the latest block in the increasing segment of C_o such that $r_{p_j} + r_{o_{i_j}} \leq 1$. We denote this alignment operation by align(j, i_j). An example is depicted in Figure 4(b,c). To make the presentation uniform, we assume the existence of a zero block for both sequences, with $l_{p_0} = l_{o_0} = 0$. Our improved basic step algorithm for bitonic C_o, termed BITONIC-FINDMIN, is depicted in Figure 5.

The time complexity of algorithm BITONIC-FINDMIN is $O(k_o + k_p \cdot \log k_o)$. The first term comes from the preprocessing step for C_o. The second term is based on the observation that both the partial sums vector s_j and the bandwidth requirements vector for the increasing segment of C_o are *sorted*, which means that the "find the least/largest index s.t. <condition>" steps of BITONIC-FINDMIN can be performed in time $O(\log k_o)$, using binary search. More elaborate search mechanisms (e.g., based on Thorup's priority queue structures [31]) can be employed to reduce the asymptotic complexity[2] of BITONIC-FINDMIN to $O(k_o + k_p \cdot \log\log k_o)$.

```
Algorithm BITONIC-FINDMIN($C_o$, $C_p$)

Input: Sequences $C_o$, $C_p$ over [0,1] in run-length compressed
    form (i.e., $C_o = < (l_{o_1}, r_{o_1}), \ldots, (l_{o_{k_o}}, r_{o_{k_o}}) >$ and
    $C_p = < (l_{p_1}, r_{p_1}), \ldots, (l_{p_{k_p}}, r_{p_{k_p}}) >$). Sequence $C_o$ is
    assumed to be *bitonic*.
Output: The least $0 \leq k \leq l(C_p)$ such that $C_o$ can be validly
    superimposed over $C_p$ starting at time slot $k$.
1. Preprocess $C_o$ to obtain a "partial sums" vector
   $s_j = \sum_{m=1}^{j} l_{o_m}$ for $j = 1, \ldots, k_o$.
2. Initialize: align(0,0), $j = i = 0$, $l_{cov} = 0$.
3. while ($l_{cov} < l(C_o)$) do
   3.1. Find the least $i_j$ such that $s_{i_j} - l_{cov} \geq l_{p_j}$.
   3.2. To check for a collision at block $(l_{p_j}, r_{p_j})$ we distin-
        guish two cases.
        • $C_o$ is decreasing after $l_{cov}$. (We just need to check
          the first block of $C_o$ placed over $(l_{p_j}, r_{p_j})$.) If $r_{p_j} +$
          $r_{o_{i_j}} \leq 1$ (i.e., no collision) then set $l_{cov} = l_{cov} + l_{p_j}$,
          $j = j+1, i = i_j$. Else, find the largest block index $m$
          in the increasing part of $C_o$ such that $r_{o_m} \leq 1 - r_{p_j}$,
          shift $C_o$ to align$(j, m)$, and set $i = m, j = j+1$,
          $l_{cov} = s_m$.
        • $C_o$ is increasing after $l_{cov}$. If the block index $i_j$ is
          in the decreasing part of $C_o$ then check if $r_{p_j} +$
          $r_{max}(C_o) \leq 1$, otherwise check if $r_{p_j} + r_{o_{i_j}} \leq 1$. If
          the condition holds set $l_{cov} = l_{cov} + l_{p_j}$, $j = j+1$,
          $i = i_j$. Else, find the largest block index $m$ in the
          increasing part of $C_o$ such that $r_{o_m} \leq 1 - r_{p_j}$, shift
          $C_o$ to align$(j, m)$, and set $i = m$, $j = j+1$,
          $l_{cov} = s_m$.
4. Return the starting time slot for the current placement of $C_o$.
```

Figure 5: Algorithm BITONIC-FINDMIN

3.2 A List-Scheduling Algorithm

We present a heuristic algorithm that uses the basic packing step described in the previous section in the manner prescribed by Graham's greedy list-scheduling strategy for multiprocessor scheduling [18]. The operation of our heuristic, termed $\mathcal{LS}$, is as follows. Let L be a list of the composite object sequences to be scheduled. At each step,

[2]Note, however, that the practicality of such search structures for the small domains encountered in this paper is questionable [23].

the $\mathcal{LS}$ algorithm takes the next object from L and (using FINDMIN or BITONIC-FINDMIN) places it at the earliest possible start point based on the current schedule. Note that this rule is identical to Graham's list-scheduling rule for an m-processor system, when all objects have a constant bandwidth requirement of $1/m$ throughout their duration.

Unfortunately, this simple list-scheduling rule does not offer a good guaranteed worst-case bound on the suboptimality of the obtained schedule. Even for the special case of bitonic objects, it is easy to construct examples where the makespan of the list schedule is $\Omega(\min\{l, |L|\})$ times the optimal makespan, where $|L|$ is the number of objects and l is the length (in rounds) of an object sequence. One such bad example for $\mathcal{LS}$ is depicted in Figure 6. Note that as the example object sequences become more "peaked", the behavior of $\mathcal{LS}$ compared to the optimal schedule becomes even worse. However, even for this bad example, $\mathcal{LS}$ will behave significantly better than MBR scheduling, which would not allow *any* overlap between consecutive "columns" in the schedule.

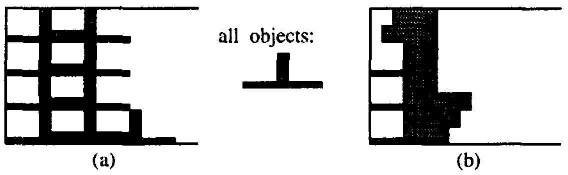


Figure 6: A "bad" example for $\mathcal{LS}$: (a) Schedule produced by $\mathcal{LS}$. (b) Optimal schedule.

Thus, in the worst case, the behavior of $\mathcal{LS}$ can be arbitrarily bad compared to the optimal schedule. Furthermore, note that since all the objects in the example of Figure 6 are *identical*, ordering the list L in any particular order (e.g., by decreasing object "height") will not help worst-case behavior. Assuming the maximum resource requirements of objects (i.e., $r_{max}(C_i)$'s) to be bounded by some small constant (a reasonable assumption for large-scale CM servers) also does not help, as long as the objects are sufficiently "peaky". However, as the following theorem shows, the situation is much better when the object sequences in L are appropriately constrained. As with all theoretical results presented here, Theorem 3.1 is stated without proof due to space constraints. The details can be found in the full version of this paper [17].

Theorem 3.1 Let L be a list of *monotonically non-increasing* composite object sequences C_i and assume that $r_{max}(C_i) \leq \lambda < 1$, for each i. Also, let $l_{max}(L) = \max_i\{l(C_i)\}$ and $V(L) = \sum_i V(C_i)$ (i.e., the total volume in L), and let $T_{OPT}(L)$ be the makespan of the optimal schedule for L. Then, the makespan returned by $\mathcal{LS}$, $T_{\mathcal{LS}}(L)$, satisfies the inequality

$$T_{\mathcal{LS}}(L) \leq \frac{V(L)}{1-\lambda} + l_{max}(L) \leq \left(1 + \frac{1}{1-\lambda}\right) \cdot T_{OPT}(L).$$

□

It is easy to see that a slightly modified version of $\mathcal{LS}$ can guarantee the same worst-case performance bound for any list of monotonically *non-decreasing* objects, as well. The basic idea is to "time-reverse" each object sequence and schedule these reversed (non-increasing) sequences using $\mathcal{LS}$. Of course, the schedule obtained is then played out in reverse (i.e., from end to start). We can combine these observations with algorithm $\mathcal{LS}$ to obtain the following simple strategy, termed Monotonic $\mathcal{LS}$ ($\mathcal{MLS}$), for scheduling *monotonic* objects. First, schedule all non-increasing sequences using $\mathcal{LS}$. Second, reverse all remaining objects and schedule them using $\mathcal{LS}$. Third, concatenate the two schedules. It is easy to prove the following corollary.

Corollary 3.1 For any list L of *monotonic* composite objects C_i with $r_{max}(C_i) \leq \lambda < 1$, for each i, algorithm $\mathcal{MLS}$ guarantees a makespan $T_{\mathcal{MLS}}(L)$ such that

$$T_{\mathcal{MLS}}(L) \leq \frac{V(L)}{1-\lambda} + 2 \cdot l_{max}(L) \leq \left(2 + \frac{1}{1-\lambda}\right) \cdot T_{OPT}(L).$$

□

With respect to the time complexity of our list-scheduling algorithms, it is easy to see that the decisive factor is the complexity of the basic packing step discussed in Section 3.1. Using the FINDMIN algorithm for general object sequences implies an overall time complexity of $O(N^2)$, where $N = \sum_{C_i \in L} n_i$ is the total number of streams used in L. If all the sequences are bitonic or monotonic, BITONIC-FINDMIN can be used giving an overall time complexity of $O(N \cdot |L| \cdot \log \frac{N}{|L|})$. (Note that that the number of composite objects $|L|$ is smaller than the number of streams N and the average number of streams per object $\frac{N}{|L|}$ is typically a small constant.)

3.3 Improving over MBRs: Monotonic Covers

Informally, we define a *monotonic cover* of a composite object C_i as another composite object $\hat{C}_i$ that (a) is monotonic, and (b) completely "covers" C_i at any point in time. More formally:

Definition 3.1 A *monotonic cover* for $C_i = < (l_{i_1}, r_{i_1}), \ldots, (l_{i_{k_i}}, r_{i_{k_i}}) >$ is an object $\hat{C}_i = < (l_{i_1}, \hat{r}_{i_1}), \ldots, (l_{i_{k_i}}, \hat{r}_{i_{k_i}}) >$ such that $\hat{r}_{i_j} \geq r_{i_j}$ for each j and $\hat{r}_{i_j} \geq \hat{r}_{i_{j+1}}$ for $j = 1, \ldots, k_i - 1$ *or* $\hat{r}_{i_j} \leq \hat{r}_{i_{j+1}}$ for $j = 1, \ldots, k_i - 1$.

Corollary 3.1 suggests an immediate improvement over the simplistic MBR assumption for scheduling composite multimedia objects: *Instead of using the MBR of an object C_i, use a minimal monotonic cover of C_i, that is, a monotonic cover that minimizes the extra volume.* List-scheduling the monotonic covers of composite objects (using $\mathcal{MLS}$) is an attractive option because of the following two reasons. First, compared to scheduling the object sequences unchanged, using the covers implies a simpler placement algorithm (a special case of BITONIC-FINDMIN), with reduced time complexity. Second, compared to using MBRs, minimal monotonic covers can significantly reduce the amount of wasted volume in the cover. (Note that a MBR is itself a trivial monotonic cover.) For example, in the case of a bitonic object, a minimal monotonic cover wastes at most half the volume that would be wasted in an MBR cover. Also note that the minimal monotonic cover of a compressed composite object sequence can be easily computed in linear time.

3.4 Utilizing Server Memory: Stream Sliding

A different way of employing our near-optimality results for list-scheduling in the case of monotonic objects is through the use of the *stream sliding* techniques described in Section 2.2. The idea is to use extra server memory to turn non-monotonic object sequences into monotonic ones. In this section, we attack the problem of efficiently utilizing server memory to make composite object sequences monotonic. More specifically, we concentrate on the use of stream *upsliding* to convert an object sequence to a non-increasing sequence *with minimal extra memory*. Our techniques are also applicable to the dual problem (i.e., making sequences non-decreasing by stream downsliding). However, as we discussed in Section 2.2, downsliding introduces latencies into the schedule and should therefore be avoided. Possible techniques that combine both upsliding and downsliding are left as an open problem for future work.

A straightforward way of making an object sequence non-increasing is to simply upslide all streams with a non-zero lag to the beginning of the composite object (Figure 7(a,b)). Given an object C_i, this naive approach will obviously require a total memory of $B \cdot \sum_{j=2}^{n_i} lt_{i_j} \cdot r_{i_j}$, where we define $lt_{i_j} = \min\{l(X_{i_j}), t_{i_j}\}$ (Section 2.2). However, we might be able to do significantly better than that. The idea is that, depending on the object's structure, it may be possible for some stream to "shield" the starting edge of another stream, without requiring the second stream to be upslided all the way to the beginning of the object. This is depicted graphically in Figure 7(c). Note that the use of such clever upsliding methods not only reduces the amount of memory required for making the object monotonic, but it also reduces the maximum bandwidth requirement of the object (i.e., $r_{max}(C_i)$). This is obviously important since it implies smaller λ's in Theorem 3.1 and, consequently, better worst-case performance guarantees for $\mathcal{LS}$.

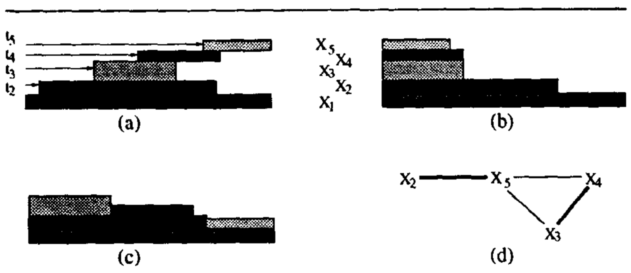


Figure 7: (a)A 5-ary composite object C. (b) "Naive" stream upsliding. (c) "Clever" stream upsliding. (d) The *shield graph* of C. (Thick lines indicate the matching used in (c).)

Unfortunately, as the following theorem shows, this problem of optimal (i.e., minimum memory) stream upsliding is $\mathcal{NP}$-hard. This result can be proved using the observation that each stream can "shield" multiple streams to give a reduction from **PARTITION** [15].

Theorem 3.2 Given a composite object sequence C_i, determining an *optimal (i.e., minimum memory)* sequence of stream upslides to make C_i non-increasing is $\mathcal{NP}$-hard. $\square$

Given the above intractability result, we now propose a simple heuristic strategy for improving upon the naive "upslide everything" method. Our solution is based on the following definition that simply states the above observations in a more formal manner.

Definition 3.2 Consider a composite object C_i. We say that stream X_{i_j} can *shield* stream X_{i_k} ($k \neq j$) if and only if $l(X_{i_j}) \leq t_{i_k}$ and $r(X_{i_j}) \geq r(X_{i_k})$. The *benefit* of shielding X_{i_k} by X_{i_j} is defined as:

$$b(j,k) = \begin{cases} B \cdot r(X_{i_k}) \cdot \left(lt_{i_k} - \min\{l(X_{i_k}), t_{i_k} - l(X_{i_j})\}\right), \\ \qquad \text{if } X_{i_j} \text{ can shield } X_{i_k} \\ 0 \qquad , \text{ otherwise} \end{cases}$$

The intuition behind Definition 3.2 can be seen from Figure 7. The benefit $b(j,k)$ is exactly the gain in server memory (compared to the naive "upslide everything" solution) by using X_{i_j} to shield X_{i_k}. Our heuristic strategy for stream upsliding uses a *edge-weighted graph* representation of the "can shield" relationships in a composite object C_i. Specifically, we define the *shield graph of* C_i, $SG(C_i)$, as an undirected graph with nodes corresponding to the streams X_{i_j} of C_i and an edge e_{jk} between node X_{i_j} and X_{i_k} if and only if X_{i_j} can shield X_{i_k} *or* X_{i_k} can shield X_{i_j}. The *weight* $w()$ of an edge is defined as the maximum gain in memory (compared to the naive approach) that can result by utilizing that edge. More formally, we define $w(e_{jk}) = \max\{\, b(j,k)\, ,\, b(k,j)\, \}$.

Our heuristic method for stream upsliding builds the shield graph $SG(C_I)$ for object C_i and determines a *maximum weighted matching* M on $SG(C_I)$. Essentially, this matching M is a collection of node-disjoint edges of $SG(C_I)$ with a maximum total weight $w(M) = \sum_{e_{jk} \in M} w((e_{jk})$. The maximum weighted matching problem for $SG(C_I)$ can be solved in time $O(n_i^3)$, where n_i is the number of streams in C_i [26]. The edges in M determine the set of "stream shieldings" that will be used in our approximate upsliding solution. Furthermore, by our edge weight definitions, it is easy to see that the the total amount of memory that will be used equals exactly $B \cdot \left(\sum_{j=2}^{n_i} lt_{i_j} \cdot r_{i_j}\right) - w(M)$, that is, the memory required by the naive approach minus the weight of the matching.

3.5 Local Improvements to $\mathcal{LS}$: List-Scheduling with Backtracking ($\mathcal{LSB}$)

The problem with the stream sliding approach outlined in the previous section is that it may often require large amounts of memory per object that the server simply cannot afford. In such cases, we are still faced with the general sequence packing problem. The results of Section 3.2

indicate that using a simple list-scheduling approach for general object sequences can result in arbitrarily bad sequence packings, leading to severe underutilization of the server's bandwidth resources. The main problem of $\mathcal{LS}$ is that by making greedy (i.e., earliest start time) decisions on the placement of objects at each step, it may end up with very "sparse" sequence packings (i.e., schedules with very poor density). This is clearly indicated in the example of Figure 6. Thus, it appears that a better scheduling rule would be, instead of trying to minimize the start time of the new object in the current schedule, try to *maximize the density* of the final, combined sequence. However, maximizing density alone also does not suffice. Returning to the example of Figure 6, it is fairly easy to see that the placement that maximizes the density of the final object is one that simply juxtaposes all the object peaks (i.e., never places one peak on top of another). Since our goal is to minimize the overall schedule length, this is not satisfactory. Instead of trying to maximize density alone, the scheduler should also make sure that the *entire* bandwidth capacity of the server (i.e., height of the bin) is utilized. (Note that using the server's bandwidth capacity instead of r_{max} to define object density does not help; since the bandwidth is fixed, placing objects to maximize this new density is equivalent to trying to maximize their overlap with the current schedule, i.e., the $\mathcal{LS}$ rule.)

In this section, we propose a novel family of scheduling heuristics for sequence packing, termed *list-scheduling with k-object backtracking* ($\mathcal{LSB}(k)$). Informally, these new algorithms try to improve upon simple $\mathcal{LS}$ by occasional local improvements to the schedule. More specifically, the operation of $\mathcal{LSB}(k)$ is as follows. The algorithm schedules objects using simple $\mathcal{LS}$, as long as the incoming objects can be placed in the current schedule without causing the length of the schedule to increase. When placing a new object results in an increase in the makespan, $\mathcal{LSB}(k)$ tries to locally improve the density of the schedule and check if this results in a better schedule. This is done in four steps. First, the last k objects scheduled are removed from the current schedule. Second, these k object sequences are combined into a single sequence in a manner that tries to maximize the density of of the resulting sequence. Third, the "combined" sequence is placed in the schedule using simple $\mathcal{LS}$. Finally, the length of this new schedule is compared to that of the original schedule, and the shorter of the two is retained. The second step in the above procedure can be performed using a slightly modified version of FINDMIN (Figure 3). The main idea is to maintain some additional state with each interval of candidate start times (step 1) and update this state accordingly when taking interval intersections (step 2). We term the resulting basic step algorithm FINDMIN-D. Similar modifications can also be defined for the BITONIC-FINDMIN algorithm for bitonic object sequences C_o. The complete $\mathcal{LSB}(k)$ algorithm is depicted in Figure 8.

Figure 9 shows the operation of the $\mathcal{LSB}(3)$ algorithm on our "bad" example for $\mathcal{LS}$ (Figure 6). More specifically, Figure 9(a) shows the schedule after the placement of the 5th object, at which point the algorithm is first forced

```
Algorithm LSB( k , L)
Input: A list of composite object sequences
       $L = < C_1, \ldots, C_n >$ and a backtracking parameter $k$.
Output: A valid packing of the object sequences in $L$.
1. Set $T_{curr} = 0$ , $C_p = \emptyset$.
2. For $i = 1$ to $n$ do
   2.1. Schedule $C_i$ at time slot FINDMIN($C_i, C_p$). Let
        $T'_{curr}$ be the length of the resulting schedule.
   2.2. If $T'_{curr} = T_{curr}$ continue. Else, do the following.
        2.2.1. Remove $C_i$ and the last $k-1$ objects from $C_p$.
               (If $k$ exceeds the number of objects in $C_p$, remove all objects from $C_p$.)
        2.2.2. Set $C = \emptyset$ and schedule each object $C_j$ removed
               from $C_p$ using FINDMIN – D($C_j, C$).
        2.2.3. Schedule $C$ over the remaider of $C_p$ using
               FINDMIN($C_i, C_p$). Let $T''_{curr}$ be the length of
               the resulting schedule.
        2.2.4. If $T'_{curr} \leq T''_{curr}$, set $T_{curr} = T'_{curr}$, restore
               the original $C_p$, and continue with the next $C_i$.
               Else, leave $C_p$ as is, set $T_{curr} = T''_{curr}$, record
               that $C$ is now a single object, and continue with
               the next $C_i$.
```

Figure 8: Algorithm $\mathcal{LSB}$

to backtrack, trying to locally improve schedule density. The result of the improved density packing (after using FINDMIN-D to combine the last three objects) is depicted in Figure 9(b). Since this schedule is shorter than the one in Figure 9(a), it is retained and $\mathcal{LSB}(3)$ goes back to using $\mathcal{LS}$. Figure 9(c) shows the schedule after $\mathcal{LS}$ places the 6th and 7th objects. Finally, Figure 9(d) shows the final schedule obtained by $\mathcal{LSB}(3)$, which is in fact the *optimal* schedule. Further, note that $\mathcal{LSB}(3)$ had to backtrack only once during the whole scheduling process.

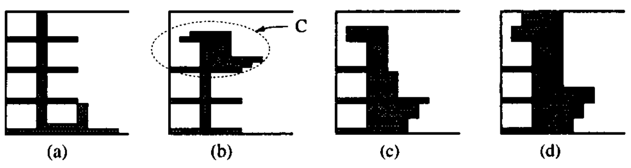


Figure 9: $\mathcal{LSB}(3)$ in action: (a)The point of the first backtracking. (b) The locally improved schedule. (c) Placing the next two objects. (d) The final (optimal) schedule.

The extra effort involved in backtracking to improve schedule density translates to increased time complexity for $\mathcal{LSB}(k)$ compared to simple $\mathcal{LS}$. Specifically, the complexity of of $\mathcal{LSB}(k)$ can be shown to be $O(|L| \cdot N^2)$, where $|L|$ is the number of objects to be scheduled and N is the total number of component streams. This, of course, assuming general object sequences that cannot employ the more efficient basic step algorithms.

4 Experimental Study

In this section, we present the results of several experiments we have conducted in order to compare the average-case performance of our composite object scheduling algorithms with that of (a) schedulers based on the MBR simplification; and, (b) the optimal schedule. Given the increased complexity of our algorithms compared to simple MBR packing, another interesting issue is the cost/benefit tradeoff involved in choosing a more elaborate scheduler. We start by presenting our experimental testbed and methodology.

4.1 Experimental Testbed

We have experimented with the following algorithms:

- $\mathcal{LS}$: Greedy, list-based scheduling of composite multimedia objects.
- $\mathcal{MBR}(FFDH)$: Level-based scheduling of composite multimedia objects using the MBR simplification and the First-Fit-Decreasing-Height (FFDH) rectangle packing method of Coffman et al. [11].

We selected the level-based FFDH rectangle packing algorithm since it is known to be one the best-performing rectangle packing methods, both in theory and in practice [11, 9, 5]. The average performance of $\mathcal{LS}$ was compared to that of $\mathcal{MBR}(FFDH)$ in order to understand the potential performance benefits of using more clever scheduling techniques to avoid the MBR simplification. We also compared the performance of the two algorithms to a *lower bound* on the response time of the optimal execution schedule. This lower bound ($LBOUND$) was estimated using the formula $LBOUND = \max\{\, l_{max}(L), V(L)\,\}$, where L is the list of objects to be scheduled, $l_{max}(L)$ is the maximum object length in L, and $V(L)$ is the total volume of all objects in L.

We experimented with randomly generated composite objects, obtained with the following procedure. First, the length and bandwidth requirement of the first (i.e., with zero lag) stream of the object was selected randomly from a set of possible lengths and rates (see Table 2). Second, the number of additional component streams was chosen randomly between 0 and 7. Third, for each of the additional streams a length and bandwidth demand was again randomly selected, and a starting point (i.e., lag) was randomly chosen across the length of the first stream. This scheme ensures that our objects are *continuous*, i.e., they have no gaps in the presentation. Note that the MBR assumption is *particularly bad* for objects with gaps, whereas our $\mathcal{LS}$ algorithm can handle these bandwidth gaps and use them effectively to schedule other objects. Thus, we expect $\mathcal{LS}$ to outperform MBR-schemes by an even larger margin when non-continuous objects are allowed. Other than the continuity restriction, note that the "shape" of the object sequences obtained with the above procedure is completely general; that is, it is not constrained to be bitonic, monotonic, etc.

The number of objects to be scheduled varied between 400 and 1400, and the server bandwidth ranged between 40 and 400 Megabits per second (Mbps). For each choice of the number of objects, ten different object lists L were generated randomly using the procedure described above for each object. We used two performance metrics in our study of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$: (1) the *average response time* of the schedules produced by the two algorithms over all lists of the same size; and, (2) the *average performance ratio* defined as the response time of the schedules produced by the algorithms divided by the corresponding lower bound and averaged over all lists of the same size. (The results presented in the next section are indicative of the results obtained for all values of the number of objects and server bandwidth.)

In all experiments, stream bandwidth demands were chosen from a discrete set of choices, ranging from 62.5 Kbps (e.g., low-quality audio) to 5 Mbps (e.g., MPEG-2 quality video). Similarly, the choice of stream lengths ranged between 10 min (e.g., a short audio clip) and 5 hrs (e.g., a long documentary). Table 2 summarizes the parameter settings used in our experiments.

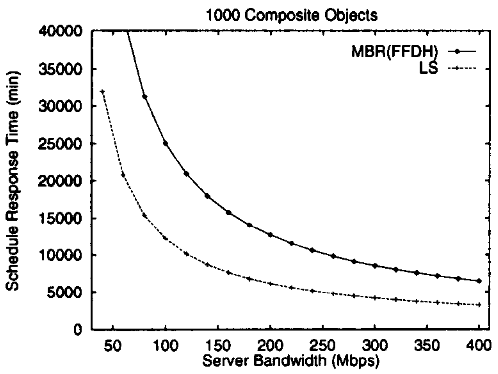
Table 2: Experimental Parameter Settings

Experimental Parameter	Value
Aggregate Server Disk Bandwidth (in Megabits per sec – Mbps)	40Mbps – 400Mbps
Number of Composite Objects	400 – 1400
Number of Streams per Object	1 – 8
Set of Possible Stream Lengths (in min)	{ 10 , 20 , 30, 60, 90, 120 , 180, 240, 300 }
Set of Possible Stream Bandwidth Requirements (in Mbps)	{ 0.0625 , 0.125 , 1 , 1.5 , 2 , 3 , 4 , 5 }

4.2 Experimental Results

Figure 10(a) depicts the average response time (in minutes) of the schedules produced by $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 1000 random composite objects. Our numbers show that $\mathcal{LS}$ consistently outperformed $\mathcal{MBR}(FFDH)$ over the entire range of available server bandwidth, offering relative improvements in the range of 50%–55%. That is, $\mathcal{LS}$ managed to cut down the schedule response time to less than half of that obtained by $\mathcal{MBR}(FFDH)$. The average density of the composite objects created during this experimental run was 0.461448, i.e., on the average, more than half of an object's MBR was "empty". Thus, although $\mathcal{MBR}(FFDH)$ does a very good job of packing the given rectangles, it is still bounded by the inherent inefficiency of the MBR simplification. On the other hand, $\mathcal{LS}$ takes advantage of the object shapes and irregularities to achieve better densities in the final schedule and, consequently, improved schedule response times.

The average performance ratios of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ obtained over the same experimental run (i.e., with 1000 composite objects) are shown in Figure 10(b). Figure 11(a) shows the average performance ratios of the two algorithms as a function of the number of objects for a fixed amount of server bandwidth (200Mbps). Note that the performance of $\mathcal{LS}$ is consistently within less



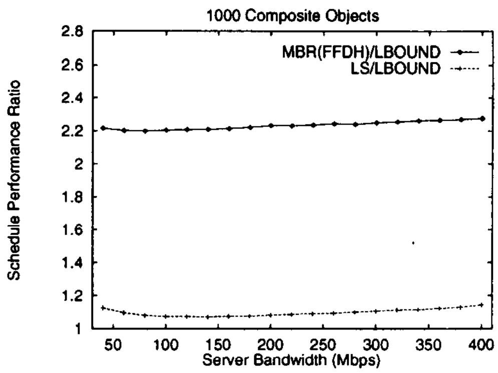


Figure 10: (a) Average schedule response times obtained by $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 1000 objects. (b) Average performance ratios of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 1000 objects.

than 15% of the lower bound on the optimal response time. Thus, even though we have shown that simple $\mathcal{LS}$ can be arbitrarily bad under certain worst-case scenarios, our results show that it offers excellent average-case behavior (for randomly generated object sequences). Furthermore, since $\mathcal{LS}$ is so close to optimal, the margin of possible improvement becomes very limited. Even if more complicated schemes like $\mathcal{LSB}$ could offer some improvement on the average over $\mathcal{LS}$, the potential benefit certainly does not seem to warrant the extra complexity. It is still possible, however, that small local perturbations on the greedy $\mathcal{LS}$ schedule, like the ones performed by $\mathcal{LSB}(k)$ with a *small* backtracking parameter k, or stream sliding methods could prove useful to avoid "bad" scenarios. Such scenarios could occur, for example, when object shapes are not completely random. We intend to investigate this issue in our future experimental work.

Finally, the table in Figure 11(b) shows the running times of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for scheduling a list of 1000 composite objects. These times were recorded on a 100Mhz SUN SPARCstation. As expected, $\mathcal{MBR}(FFDH)$ is significantly faster than $\mathcal{LS}$, since its complexity is only $O(|L| \cdot \log |L|)$, where $|L|$ is the number of composite objects to be scheduled. On the other hand, $\mathcal{LS}$ is still fast enough for all practical purposes (the average scheduling time per object is a few milliseconds) and, as our results have shown, offers dramatically improved schedules compared to $\mathcal{MBR}(FFDH)$.

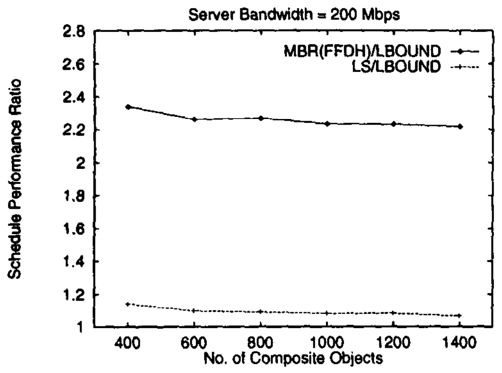
5 Ongoing Work: Stream Sharing

Until now, we have implicitly assumed that the composite multimedia objects to be scheduled are *disjoint*, in the sense that their component streams correspond to different data objects in the underlying repository. However, it is quite possible for distinct composite objects to have one or more components in common. Examples of such non-disjoint composite objects range from simple movie presentations, where the same video needs to be displayed with different soundtracks (e.g., in different languages), to complex news stories authored by different users, that can share a number of daily event clips.

In the presence of such common components, it is possible that exploiting *stream sharing* can lead to better schedules. The basic idea is that by appropriately scheduling non-disjoint composite objects, the streams delivering their common component(s) can be shared by all the composite object presentations. Clearly, stream sharing can reduce the aggregate resource requirements of a set of non-disjoint objects and it is easy to construct examples for which exploiting stream sharing can drastically improve the response time of a presentation schedule. On the other hand, stream sharing possibilities also increase the complexity of the relevant scheduling problems. Even simple cases of the problem (e.g., when all streams and composite objects are of unit length) give rise to hard scheduling problems that have not been addressed in the scheduling literature [8]. The problem becomes even more challenging when extra memory is available, since stream sliding and caching techniques can be used to increase the possibilities for stream sharing across composite objects. Finally, note that our stream sharing problem also appears to be related to the problem of exploiting common subexpressions during the simultaneous optimization of multiple queries [27]. However, the use of a schedule makespan optimization metric (instead of total plan cost) makes our problem significantly harder to formulate and solve.

6 Conclusions and Future Work

Effective resource scheduling for composite multimedia objects is a crucial requirement for next generation multimedia database systems. Despite the importance of the problem, the complexity of the relevant task scheduling models has limited prior research to very specific subproblems. Furthermore, today's systems typically employ worst-case (i.e., MBR) assumptions that can lead to severe wastage of precious server resources. In this paper, we have presented a novel *sequence packing* formulation of the composite object scheduling problem and we have proposed novel efficient algorithms drawing on techniques from pattern matching and multiprocessor scheduling. More specifically, we have developed efficient "basic step" methods for combining two object sequences into a single, combined sequence and we have incorporated these methods within: (1) a simple, greedy scheduler base on Graham's list-scheduling paradigm ($\mathcal{LS}$); and, (2) a more complex scheduler ($\mathcal{LSB}$) that tries to improve upon sim-



Time to schedule 1000 objects (in sec)

Bandwidth	MBR(FFDH)	LS
100	0.016	9.041
200	0.012	6.233
300	0.010	4.709
400	0.009	3.760

Figure 11: (a) Average performance ratios of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 200 Mbps of server bandwidth. (b) Running times for $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$.

ple $\mathcal{LS}$ by occasional local backtracking. We have shown that although simple list-scheduling schemes are provably near-optimal for monotonic object sequences, they exhibit poor worst-case performance for general object sequences. It is exactly this worst-case behavior that $\mathcal{LSB}$ has been designed to avoid. On the other hand, our experimental results with randomly generated objects have shown that simple $\mathcal{LS}$ offers excellent average-case performance compared to both an MBR-based approach and the optimal solution.

Besides our ongoing work on stream sharing, our future research plans include: (1) extending our single-resource environment (i.e., disk bandwidth) for the sequence packing problem to a multi-resource setting [28, 16], and (2) exploiting flexible Quality of Service (QoS) specifications for streams to obtain more effective schedules.

References

[1] J.F. Allen. "Maintaining Knowledge About Temporal Intervals". *Comm. of the ACM*, 26(11):832–843, 1983.

[2] A. Amir and M. Farach. "Efficient 2-dimensional Approximate Matching of Non-rectangular Figures". In *Proc. of the 2nd Annual ACM-SIAM Symp. on Discrete Algorithms*, January 1991.

[3] A. Apostolico and Z. Galil, eds. *"Pattern Matching Algorithms"*. Oxford University Press, 1997.

[4] B.S. Baker, E.G. Coffman, Jr., and R.L. Rivest. "Orthogonal Packings in Two Dimensions". *SIAM Journal on Computing*, 9(4):846–855, 1980.

[5] B.S. Baker and J.S. Schwarz. "Shelf Algorithms for Two-Dimensional Packing Problems". *SIAM Journal on Computing*, 12(3):508–525, 1983.

[6] S. Chakrabarti, C.A. Phillips, A.S. Schulz, D.B. Shmoys, C. Stein, and J. Wein. "Improved Scheduling Algorithms for Minsum Criteria". In *Proc. of the 23rd Intl. Colloquium on Automata, Languages, and Programming (ICALP'96)*, July 1996.

[7] S.Chaudhuri, S. Ghandeharizadeh, and C. Shahabi. "Avoiding Retrieval Contention for Composite Multimedia Objects". In *Proc. of the 21st Intl. Conf. on Very Large Data Bases*, September 1995.

[8] E.G. Coffman. Personal Communication, February 1998.

[9] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. "Approximation Algorithms for Bin-Packing – An Updated Survey". In *"Algorithm Design for Computing System Design"*, pages 49–106. Springer-Verlag, New York, 1984.

[10] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. "Approximation Algorithms for Bin-Packing: A Survey". In *"Approximation Algorithms for NP-Hard Problems", D. Hochbaum (Ed.)*, pages 46–93. PWS Publishing, Boston, 1996.

[11] E.G. Coffman, Jr., M.R. Garey, D.S. Johnson, and R.E. Tarjan. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms". *SIAM Journal on Computing*, 9(4):808–826, 1980.

[12] M. Farach. Personal Communication, December 1997.

[13] M.R. Garey and R.L. Graham. "Bounds for Multiprocessor Scheduling with Resource Constraints". *SIAM Journal on Computing*, 4(2):187–200, 1975.

[14] M.R. Garey, R.L. Graham, D.S. Johnson, and A.C. Yao. "Resource Constrained Scheduling as Generalized Bin Packing". *Journal of Combinatorial Theory (A)*, 21:257–298, 1976.

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

[16] M.N. Garofalakis and Y.E. Ioannidis. "Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources". In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, August 1997.

[17] M.N. Garofalakis, Y.E. Ioannidis, and B. Özden. "Resource Scheduling for Composite Multimedia Objects". Tech. Memorandum BL0112330-980225-03TM, Bell Laboratories, February 1998.

[18] R.L. Graham. "Bounds on Multiprocessing Timing Anomalies". *SIAM Journal on Computing*, 17(2):416–429, 1969.

[19] L.A. Hall, A.S. Schulz, D.B. Shmoys, and J. Wein. "Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms". *Mathematics of Operations Research*, 22:513–544, 1997.

[20] D.E. Knuth, J.H. Morris, and V.R. Pratt. "Fast Pattern Matching in Strings". *SIAM Journal on Computing*, 6(2):323–350, 1977.

[21] L.T. Kou and G. Markowsky. "Multidimensional Bin Packing Algorithms". *IBM Journal of Research and Development*, September 1977.

[22] C. Martin, P.S. Narayanan, B. Özden, R. Rastogi, and A. Silberschatz. "The *Fellini* Multimedia Storage Server". In *"Multimedia Information Storage and Management", S.M. Chung (Ed.)*. Kluwer Academic Publishers, 1996.

[23] S. Muthukrishnan. Personal Communication, January 1998.

[24] S. Muthukrishnan and K. Palem. "Non-standard Stringology: Algorithms and Complexity". In *Proc. of the 26th Annual ACM Symp. on the Theory of Computing*, May 1994.

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

[26] C. Papadimitriou and K. Steiglitz. *"Combinatorial Optimization: Algorithms and Complexity"*. Prentice Hall, Inc., 1982.

[27] T.K. Sellis. "Multiple-Query Optimization". *ACM Trans. on Database Systems*, 13(1):23–52, 1988.

[28] H. Shachnai and J.J. Turek. "Multiresource Malleable Task Scheduling". Unpublished Manuscript, July 1994.

[29] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri. "On Scheduling Atomic and Composite Multimedia Objects". Tech. Report USC-CS-95-622, Univ. of Southern California, 1995. (To appear in *IEEE Trans. on Knowledge and Data Engineering.*).

[30] D.B. Shmoys, J. Wein, and D.P. Williamson. "Scheduling Parallel Machines On-line". *SIAM Journal on Computing*, 24(6):1313–1331, 1995.

[31] M. Thorup. "On RAM Priority Queues". In *Proc. of the 7th Annual ACM-SIAM Symp. on Discrete Algorithms*, January 1996.

[32] J. Turek, W. Ludwig, J.L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P.S. Yu. "Scheduling Parallelizable Tasks to Minimize Average Response Time". In *Proc. of the 6th Annual ACM Symp. on Parallel Algorithms and Architectures*, June 1994.