On Scheduling Atomic and Composite Multimedia Objects

Cyrus Shahabi, Shahram Ghandeharizadeh  
Computer Science Department  
University of Southern California  
Los Angeles, California 90089

[shahabi, shahram]@perspolis.usc.edu

Surajit Chaudhuri  
Hewlett-Packard Labs  
1501 Page Mill Road  
Palo Alto, CA 94304

surajit@hplsc.hpl.hp.com

Abstract

In multi-user multimedia information systems (e.g., video-on-demand, news-editing) the policy employed to activate queued requests has a significant impact on the average startup latency observed by the users. In this paper, we view the retrieval of an object as a task and study the problem of scheduling such tasks. In this scheduling problem, tasks are IO-bound and each may utilize multiple disks. Each task acquires and releases disks in a regular manner based on the layout of its referenced object on the disks. In addition, there might be temporal relationships among multiple tasks that constitute a composite task. While previous studies have focused on both scheduling subtasks localized to a single disk (e.g., GSS [YCK92]) and determining the regular pattern of disk utilization per task (e.g., striping [BGMJ94]), the method for selecting a task for activation has received little attention.

In this study, we formalize a class of task scheduling problems that arise in a large class of multimedia applications. We show that task scheduling problems are intractable (NP-hard), justifying the use of heuristics. We tailor some of the well known scheduling heuristics, studied in other contexts, and by means of a simulation study conclude that one of the heuristics (FFD+ECF) is superior to the other alternatives. We also demonstrate that the average latency time can be decreased further by employing buffers to resolve contention.

---

*This research was supported in part by the National Science Foundation under grants IRI-9203389, IRI-9258362 (NYI award); and CDA-9216321, and a Hewlett-Packard unrestricted cash/equipment gift.
1 Introduction

Multimedia information systems that provide service to multiple simultaneous requests are starting to become common place. In these systems, the policy employed to service requests impacts the average startup delay incurred by the users. This impact becomes significant in a heavily loaded system where servicing the tasks in their order of arrival might result in a high average startup delay, as will be shown in our experiments. This paper investigates three classes of multimedia applications and alternative policies for scheduling their requests. The application classes include:

1. **On-demand Atomic Object Retrieval**: With this class of applications, a system strives to display an object (audio or video) as soon as a user request arrives referencing the object. The envisioned video-on-demand and news-on-demand systems are examples of this application class. We formalize the scheduling problem that represents this class as *Atomic Retrieval Scheduling* (ARS) problem.

2. **Reservation-based Atomic Object Retrieval**: This class is similar to on-demand atomic object retrieval except that a user requests the display of an object at some point in the future. An example might be a video-on-demand system where the customers call to request the display of a specific movie at a specific time, e.g., Bob calls in the morning to request a movie at 8:00pm. Reservation-based retrieval is expected to be cheaper than on-demand retrieval because it enables the system to minimize the amount of resources required to service requests (using planning optimization techniques). The scheduling problem that represents this application class is termed *Augmented ARS* (*ARS*⁺).

3. **On-demand Composite Object Retrieval**: As compared to atomic objects, a composite object describes when two or more atomic objects should be displayed in a temporarily related manner [All83]. To illustrate the application of composite objects, consider the following environment. During the post-production of a movie, a sound editor accesses an archive of digitized audio clips to extend the movie with appropriate sound-effects. The editor might choose two clips from the archive: a gun-shot and a screaming sound effect. Subsequently, she authors a composite object by overlapping these two sound clips and synchronizing them with the different scenes of a presentation. During this process, she might try alternative gun-
shot or screaming clips from the repository to evaluate which combination is appropriate. To enable her to evaluate her choices immediately, the system should be able to display the composition as soon as it is authored (on-demand). The scheduling problem that represents this application class is termed *Composite Retrieval Scheduling* (CRS).

In this study, we analyze the above three scheduling problems in the context of an off-line (clairvoyant) scheduler, where the system has a complete knowledge about all the tasks released in future. Since these scheduling problems are novel, we show that they are *NP*-hard, justifying the use of heuristics. We revisit the heuristics proposed for the conventional scheduling problems and adopt them for ARS, $ARS^+$, and CRS. The adopted heuristics are efficient and fine tuned to serve as policies for activating requests in an on-line system. (An on-line system services requests as they arrive and has no knowledge of requests that might arrive in the future.)

A contribution of this paper is that it relates the three scheduling problems. We argue that $ARS^+$ is the core scheduling problem because it is a generalization of ARS and a specialization of CRS. Therefore, the on-line policies are examined for $ARS^+$ using a simulation study. We conclude that one of the policies (FFD+ECF) is superior to others. The role of buffers to reduce the average startup delay is also investigated and we show that for a low system load it can reduce the average startup delay significantly.

To put our work in perspective, we denote the retrieval of an object as a *task*. Subsequently, we consider scheduling of tasks having three levels of abstractions where level one and two have been investigated in detail and the last level is the main topic covered by this study and has not been studied before to the best of our knowledge. Level one investigates how a single continuous display should utilize multiple disk drives [TPBG93, BGMJ94, GK95]. Partitioning each object to a number of *subobjects*, this depends on the placement of the subobjects of an object across the disks. In this paper we assume a round-robin assignment of the subobjects of an object to the disks, starting with an arbitrary disk [BGMJ94, GK95] (see Section 2.2). Level two determines the order that different subtasks (each subtask corresponds to the retrieval of a subobject) should be invoked on a single disk. This ordering depends on the placement of the subobjects on the disk and is done to minimize the impact of the seek time. In this study, we employ the *cycle-based* [TPBG93, YCK92, BGM95]
approach and extend it to support the continuous display of a mix of media types (see Section 2.1). Level three determines the order that a queue of pending tasks should be activated. This scheduling depends on how the tasks utilize the disks. It introduces different challenges as compared to those addressed in other multimedia studies [TPBG93, DSS94, VGG94] (see Section 5). Moreover, scheduling problem in the presence of temporal relationships among tasks is novel and has not been addressed before.

To summarize, the distinctive characteristics of the scheduling problems (ARS, ARS+, CRS) are: 1) tasks are IO-bound and not CPU-bound, 2) each task utilizes multiple disks during its lifetime, 3) each task acquires and releases disks in a regular manner, 4) the pattern that a task utilizes the disks depends on the placement of its referenced object on the disks, and 5) there might be temporal relationships among multiple tasks constituting a composite task\(^1\).

## 2 Framework

In this section, we start by extending the cycle-based approach to support the continuous display of atomic objects belonging to a mix of media type assuming a single disk hardware platform. Subsequently, we extend the hardware architecture to a multi-disk platform. Most importantly, we identify two desirable criteria that should be maintained for efficient scheduling on a multi-disk platform. Finally, we describe the continuous display of composite objects on a multi-disk hardware platform.

### 2.1 Continuous Display of a Mix of Media Types, Single Disk

When displaying an object, it must be rendered to the screen at a pre-specified bandwidth. This implies that the object must be staged from disk to memory at a pre-specified rate. If retrieved at a lower rate with no precautions (e.g., prefetching), then the display will suffer from frequent disruptions and delays, termed hiccups. Different media types may require different bandwidth

---

\(^1\)For a discussion on how the collection of the above characteristics distinguish this problem from the previous scheduling problems see Section 5.
requirements. For example, a MPEG-1 compressed video object requires a 1.5 megabit per second (Mb/s) bandwidth. A MPEG-2 compressed video object may require a bandwidth ranging from 4 to 15 Mb/s depending on the resolution used to compress data. A stereo CD quality audio object has a 1.4 Mb/s bandwidth requirement [AOG91].

Assume \( m \) media types each with a bandwidth requirement of \( c_i \). To display an object \( X \), it is partitioned into \( f \) subobjects: \( X_1, X_2, \ldots, X_f \). The size of a subobject is a function of its consumption rate and a pre-defined subobject size for media type \( i \). This size is chosen such that the display time of all subobjects are identical independent of their media type. The time to display a subobject is termed a *time interval*. Once object \( X \) is referenced, its display employs a cycle-based approach: the system stages the first subobject of \( X \) into memory during time interval and initiates its display at the beginning of the second time interval. During the second time interval, the system stages \( X_2 \) into memory and initiate its display at the beginning of the third time interval. This process is repeated until all subobjects of \( X \) have been retrieved and displayed. During the time interval that subobject \( X_f \) is displayed, no subobjects are retrieved on behalf of \( X \).

Given a database with a mix of media types, the display time of each subobject of the different objects must be identical in order to fixed-size time intervals (and a continuous display for a mix of displays referencing different objects). This is achieved as follows. First, objects are grouped based on their media types. Next, the system chooses media type \( i \) with subobject size \( \text{Sub}_i \) and bandwidth requirement \( c_i \) to determine the duration of a time interval. The subobject size of a media type \( j \) is chosen to satisfy the following constraint: \( \text{interval} = \frac{\text{Sub}_i}{c_j} = \frac{\text{Sub}_i}{c_j} \).

For now assume the bandwidth required to display an object is lower than the bandwidth of a single disk\(^2\). Hence, the time to retrieve subobject \( X_i \) from a disk is shorter than the duration

\(^2\)We will relax this assumption in Section 2.2.
of a time interval (i.e., the time to display subobject \(X_i\)). This enables the system to retrieve and display several objects simultaneously. For example, Fig. 1 shows the retrieval and display schedule for objects \(X\), \(Y\), and \(Z\). During the first cycle of this figure, the system reads subobjects \(W_i, X_j,\) and \(Z_k\) from disk to memory while displaying \(W_{i-1}, X_{j-1},\) and \(Z_{k-1}\). During the next cycle, the system reads subobjects \(W_{i+1}, X_{j+1},\) and \(Z_{k+1}\) and displays subobjects \(W_i, X_j,\) and \(Z_k\). Within an interval, the different subobjects can be retrieved based on an elevator scheduling policy in order to maximize the utilization of disk bandwidth [YCK92].

### 2.2 Multi-Disk Hardware Platform

Assume a multi-disk platform consisting of \(N\) disks. We partition the disks into \(D\) clusters each with \(k\) physical disks: \(D = \lceil \frac{N}{k} \rceil\) [TPBG93, GDS95, BGM95, GK95]. Next, we assign the subobjects of \(X\) to the clusters in a round-robin manner, starting with an arbitrarily chosen cluster\(^3\). Each subobject of \(X\) is declustered [GRAQ91] into \(k\) fragments, with each fragment assigned to a different disk in the cluster. For example, in Fig. 2, a system consisting of six disks is partitioned into three clusters, each consisting of two disk drives. The assignment of the subobjects of \(X\) starts with cluster 1. This subobject is declustered into two fragments: \(X_{1,0}\) and \(X_{1,1}\). The bandwidth of a cluster should always exceed the bandwidth requirements of an object in order to minimize the amount of required memory. We conceptualize the \(k\) disks in a cluster as a single logical disk because a read request activates all \(k\) disks in a cluster. **Henceforth, the term disk is used synonymously with the term cluster.** We number the disks (or clusters) from 1 to \(D\) (\(d_1, \ldots, d_D\)). To display object \(X\), the system locates the disk containing its first subobject, say disk \(d_i\). Assuming that disk \(d_i\) has sufficient bandwidth available to retrieve \(X_1\) during the current time interval, the system initiates its retrieval from \(d_i\). At the beginning of next time interval, the system initiates the display of \(X\). This is due to the cycle-based approach where the display is always one interval behind the retrieval. For the rest of this paper we will ignore this extra delay when computing the startup delays observed by the requests.

Once all the objects are laid out on the same set of disks, the new problem of **retrieval contention**

---

\(^3\)The round-robin assignment helps to speed up the scheduling (see Section 2.2.1).
arises. That is, two subobjects of two different objects might compete for the bandwidth of the same disk at a specific time interval. The reason is that both subobjects are assigned to the same disk. Due to the retrieval contention, the system might not be able to retrieve both subobjects. To avoid the possibility of hiccups, a scheduler that schedules the tasks on behalf of object retrievals should guarantee that once a task is initiated it will not collide with other tasks. In other words, their corresponding objects will not face retrieval contention. The retrieval contention is the main characteristic of our scheduling problems that distinguishes them from conventional CPU scheduling problems (see Section 5 for a detailed comparison).

2.2.1 Desirable Features of Scheduling for Fast Contention Prediction

A major characteristic of scheduling algorithms is to find the task(s) that should be initiated at each time interval rapidly, while ensuring that the scheduling will not result in a retrieval contention with the active tasks. The duration of a time interval is in the order of seconds and the scheduler should be invoked at the beginning of every time interval to predict retrieval contention. Although this prediction is CPU-bound, its duration should occupy a very small portion of a time interval because the remainder of the time interval should be utilized by the tasks to retrieve the corresponding subobjects. Assuming that contention prediction per task requires 0.33 ms and 600 tasks are examined per time interval, 10% of a 2 second time interval will be occupied by the scheduler. We discuss two criteria that enable a scheduler to predict contentions efficiently.

First, the subobjects should be assigned to disks using a regular pattern, e.g., round-robin assignment. This helps to speed up the prediction because to initiate a retrieval task it is sufficient to examine only the current interval for contention. To observe, consider Fig. 3. In this figure we made the simplifying assumptions that all objects belong to a single media type (with bandwidth requirement \( c \)), and a single disk can support the retrieval of a single subobject per time interval.
(i.e., $R_D = c$). Each box in row $u$ and column $d$ demonstrates the status of disk $d$ at time interval $u$. An empty box means that the cluster is idle, while a box with letter $X_k$ shows that the disk is busy retrieving the $k$th subobject of atomic object $X$. In Figs. 3a and 3b, a request referencing object $Y$ arrives at time interval $i$. With random assignment of the subobjects (Fig. 3a), in order to start $Y$ at $i$, the scheduler needs to examine the future intervals to guarantee that there would be no contention for the entire retrieval duration of $Y$. Assuming a two hour movie and 2 second time interval, 3600 elements should be examined for contention per task. Given a 100 MIPS CPU and 10 instructions per examination, contention prediction per task requires 0.33 ms resulting in 10% occupation of time interval when checking 600 tasks. With a regular layout (Fig. 3b), however, by knowing that the first subobject of $Y$ will not result in contention at $i$, the scheduler can guarantee that there would be no contention for the entire retrieval duration of $Y$.

Second, the scheduler should not reserve resources in advance. For example, in Fig. 3c where the objects still follow the round-robin assignment, assume a request referencing object $Y$ arrives at time $i - 1$ while disk 3 is busy until time $i + 2$. Suppose that the scheduler looks ahead and reserves disk 3 at time $i + 2$ for $Y$. Now, at time $i$, a new request arrives referencing object $Z$ and since $Z_1$ can be scheduled at $i$ with no contention, its retrieval task is initiated at $i$. However, due to the advance resource reservation for object $Y$, a collision will occur at $i + 2$. The scheduler needs to look ahead when scheduling $Z$ to prevent such a collision. However, similar to the case of random assignment, this requires the examination of several time intervals for the entire retrieval duration of $Z$.

The proposed scheduling algorithms for atomic objects respect these two criteria. However, we show that these criteria cannot be guaranteed when scheduling the retrieval of composite objects.

### 2.3 Composite Objects

We conceptualize a system that supports composite objects as consisting of three components: a collection of user interfaces, logical abstraction, and a storage manager (see Figure 4). User interfaces play an important role in providing a friendly interface to (1) access existing data to author composite objects and (2) display objects. The logical abstraction tailors the user interface
to the storage manager and is described further in Section 2.3. The focus of this study is on the storage manager. Given a display schedule, this study investigates techniques to construct retrieval schedules that satisfy the temporal constraints of a display schedule. The retrieval schedule dictates when different portions of an object should be staged from disk into memory to be available for display. If the retrieval schedule fails to satisfy the temporal constraints described by the author then the display of an object may suffer from either delays or logical errors, e.g., the display produces a gun shot either a few seconds too early or too late, the lips of an actor may no longer be synchronized with his voice, etc. These failures are collectively termed hiccups. Our objective is to construct retrieval schedules in support of a hiccup-free display.

At the logical level of abstraction (see Figure 4), a composite object is represented as a \((X, Y, j)\) indicating that the composite object consists of atomic objects \(X\) and \(Y\). The parameter \(j\) is the lag parameter: It indicates that the display of object \(Y\) should start \(j\) time interval after the display of \(X\) has started. For example, to designate a complex object where the display of \(X\) and \(Y\) must start at the same time, we will use the notation \((X, Y, 0)\). Likewise, the composite object
specification \((X, Y, 2)\) indicates that the display of \(Y\) is initiated two intervals after the display of \(X\) has started. This definition of a composite object supports the alternative temporal relationships described in [All83] (see [CGS95] for more details). This notation extends naturally to specification of composite objects that contain more than one atomic objects. A composite object containing \(n\) atomic objects can be characterized by \((n - 1)\) lag factor, e.g., \((X^1, X^n, j^2, \ldots, j^n)\) where \(j^i\) denotes the lag factor of object \(X^i\) with respect to the beginning of the display of object \(X^1\).

To simplify the discussion we assume integer values for the lag parameter (i.e., the temporal relationships are in the granularity of time interval). For more accurate synchronization such as lip-synching between a spoken voice with the movement of the speaker’s lips, real values of the lag parameter should be considered. This extension is straightforward. To illustrate, suppose time dependency between objects \(X\) and \(Y\) is defined such that the display of \(Y\) should start 2.5 seconds after the display of \(X\) starts. Assuming the duration of a time interval is one second, this time dependency at the task scheduling level can be mapped to \((X, Y, 2)\). Hence, the system can retrieve \(Y\) after 2 seconds, but employ memory to postpone \(Y\)’s display for 0.5 seconds.

With composite objects, the problem of retrieval contention becomes more severe when it occurs among the atomic objects constituting a composite object. This type of contention is termed internal contention. The formal definition of internal contention and its solution is discussed in Section 3.3.2.

The rest of this paper is organized as follows. Section 3 describes three scheduling problems (ARS, \(ARS^+\), CRS) and propose some heuristics for solving them. Section 4 evaluates the proposed heuristics using a simulation model. In Section 5, we distinguish this study from the other related studies. Our conclusions and future research directions are contained in Section 6.

### 3 Class of Scheduling Problems

To tackle the scheduling problem, first we focus on the task of scheduling retrievals of atomic objects that belong to different media types on a multi-disk/multi-user environment. We term this problem Atomic Retrieval Scheduling, ARS. Subsequently, we introduce an augmented model of
ARS, termed $ARS^+$. Finally, we focus on the task of constructing retrieval schedules for composite objects. This process is termed *Composite Retrieval Scheduling*, CRS. We show that the solutions proposed for ARS and $ARS^+$ can be extended to support CRS.

### 3.1 Atomic Retrieval Scheduling (ARS)

To motivate the ARS problem, consider a *video-on-demand* application. Each object can be considered as a movie with different bandwidth requirement. One compressed by MPEG-1 and the other by MPEG-2, or movies with different display qualities (e.g., NTSC or HDTV). Now the question is in which order a number of requests referencing these movies should be serviced so that all the customers observe a tolerable startup delay. By slight variation of ARS objectives, one can answer a number of other interesting questions, such as: *How much resources are required so that nobody observes a startup delay greater than $x$ seconds?*

With ARS, the retrieval of each object is termed a retrieval task. The ARS problem is to schedule retrieval tasks such that the total bandwidth requirement of the scheduled tasks on each disk during each interval does not exceed the bandwidth of that disk. Moreover, ARS should satisfy an optimization objective. This objective is application dependent. It might be either to: 1) minimize the average startup delay of the tasks, or 2) minimize the total duration of scheduling for a set of tasks. The major assumption is that the bottleneck resource is disk bandwidth, and there is always an idle processor to schedule the retrieval task [GGJY76]. This is a valid assumption because the performance of magnetic disk drives are restricted by their mechanical components and is not expected to improve significantly in the near future [PGK88] (performance improvements have been rated at only 7 to 10 percent annually [RW94]). In this section, we present: 1) formal mathematical model of ARS, 2) show that it is a $NP$-hard problem, and 3) investigate some heuristics for on-line scheduling algorithms.

#### 3.1.1 Defining Tasks

Let $\mathcal{T}$ be a set of tasks where each $t \in \mathcal{T}$ is a retrieval task corresponding to the retrieval of a video object. Note that if an object $X$ is referenced twice by two different requests, a different task
is assigned to each request. The time to display a subobject is defined as a *time interval* that is employed as the time unit in this study.

For each task $t \in T$, we define:

- $r(t)$: Release time of $t$, $r : T \rightarrow N$. The time that $t$'s information (i.e., the information of its referenced object) becomes available to the scheduler. This is identical to the time that a request referencing the object is submitted to the system.

- $l(t)$: Length (size) of the object referenced by $t$, $l : T \rightarrow N$. The unit is in number of subobjects.

- $c(t)$: Consumption rate of the object referenced by $t$, $0 < c(t) \leq 1$. This rate is normalized by $R_D$. Thus, $c(t) = 0.40$ means that the consumption rate of the object referenced by $t$ is 40% of $R_D$.

- $p(t)$: The disk that contains the first subobject of the object referenced by $t$, $1 \leq p(t) \leq D$.

### 3.1.2 Formal Definition of ARS

**Definition 3.1:** The problem of ARS is to find a schedule $\sigma$ (where $\sigma : T \rightarrow N$) for a set $T$, such that: 1) it minimizes the *finishing time* $w$, where $w$ is the least time at which all tasks of $T$ have been completed, and 2) satisfies the following constraints:

- $\forall t \in T$, $\sigma(t) \geq r(t)$.

- $\forall u \geq 0$, let $S(u)$ be the set of tasks which $\sigma(t) \leq u < \sigma(t) + l(t)$, then: $\forall i$, $1 \leq i \leq D$

  $\sum_{t \in S(u)} R_i(t) \leq 1.0$ where:

  $$R_i(t) = \begin{cases} 
  c(t) & \text{if } (p(t) + u - \sigma(t)) \mod D = i - 1 \\
  0.0 & \text{otherwise}
  \end{cases} \quad (1)$$

The first constraint ensures that no task is scheduled before its release time. The second constraint strives to avoid retrieval contention. It guarantees that at each time interval $u$ and for each disk $i$, the aggregate bandwidth requirement of the tasks that both employ disk $i$ and are in progress (i.e., have been initiated at or before $u$ but have not committed yet), do not exceeds the bandwidth of disk $i$. The $\mod$-function handles the round-robin utilization of disks per task. ■
Example 3.2: Suppose: \( D = 3, T = \{t_1, t_2, t_3\} \). Given the task information as:
\( t_i := (r(t_i), l(t_i), c(t_i), p(t_i)) \), then:
\[
\begin{align*}
t_1 & := (0, 5, 0.6, 1) > t_2 := (2, 5, 0.8, 2) > t_3 := (3, 5, 0.6, 1)
\end{align*}
\]

The required display schedules of these tasks as a function of time is depicted in Fig. 5a. The placement of objects \( X, V, \) and \( Z \) referenced respectively by \( t_1, t_2, \) and \( t_3 \) is shown in Fig. 6. Consider the schedule \( \sigma(t) = r(t) \) that results in the minimum \( w \) (no task can start sooner than its release time). However, this is not a feasible schedule. To illustrate, consider the following discussion:

- The schedule is:
  \[
  \begin{align*}
  \sigma(t_1) &= 0, \sigma(t_1) + l(t_1) = 5 \quad \sigma(t_2) = 2, \sigma(t_2) + l(t_2) = 7 \quad \sigma(t_3) = 3, \sigma(t_3) + l(t_3) = 8
  \end{align*}
  \]

- Iterating over the time intervals \( u \):
  - \( u = 0 \Rightarrow S(0) = \{t_1\} \). Hence,
    \[
    R_1(t_1) = 0.6 \leq 1.0 \quad R_2(t_1) = 0.0 \leq 1.0 \quad R_3(t_1) = 0.0 \leq 1.0
    \]
  - \( u = 1 \Rightarrow S(1) = \{t_1\} \). Hence,
    \[
    R_1(t_1) = 0.0 \leq 1.0 \quad R_2(t_1) = 0.6 \leq 1.0 \quad R_3(t_1) = 0.0 \leq 1.0
    \]
- \( u = 2 \Rightarrow S(2) = \{t_1, t_2\} \). Hence,
\[
R_1(t_1) + R_1(t_2) = 0.0 \leq 1.0 \quad R_2(t_1) + R_2(t_2) = 0.8 \leq 1.0 \quad R_3(t_1) + R_3(t_2) = 0.6 \leq 1.0
\]
- \( u = 3 \Rightarrow S(2) = \{t_1, t_2, t_3\} \). Hence,
\[
R_1(t_1) + R_1(t_2) + R_1(t_3) = \frac{1.2}{1} > 1.0 \\
R_2(t_1) + R_2(t_2) + R_2(t_3) = 0.0 \leq 1.0 \\
R_3(t_1) + R_3(t_2) + R_3(t_3) = 0.8 \leq 1.0
\]

- Since \( u = 3 \), the aggregate bandwidth requirement of \( t_1, t_2, \) and \( t_3 \) exceeds the bandwidth of \( d_1 \), the proposed schedule (i.e., \( \sigma(t) = r(t) \)) is not a feasible one.

Note that in this example, as well as future examples in this paper, we assume a high bandwidth requirement for the objects (i.e., large \( c(t) \)). This is done to simplify the discussions. Instead, in Section 4, we used realistic values for \( c(t) \) when executing our experiments (see Table 3).

### 3.1.3 ARS is NP-hard

ARS can be shown to be NP-hard by a reduction from Bin Packing Problem [GJ75]. The key intuition is that deciding which requests to pack together for unit disk bandwidth can be reduced to the problem of packing objects in a bin.

**Definition 3.3: Bin Packing Problem:** Given a finite set \( U \) of items and a rational size \( s(u) \in [0, 1] \) for each item \( u \in U \), find a partition of \( U \) into disjoint subsets \( U_1, \ldots, U_k \) such that the sum of the sizes of items in each \( U_i \) is no more than 1 and such that \( k \) is as small as possible.

**Theorem 3.4:** ARS is NP-hard.

**Proof:** We use a reduction from bin packing. We let the total disk bandwidth to be 1 and corresponding to every \( u \in U \), we create a task with the bandwidth requirement \( s(u) \). We can show that the problem of minimizing the finishing time for the above set of tasks is equivalent to minimizing the number of bins for the set of objects \( U \). Assume that \( k \) is the fewest number of
bins in which the objects may be packed. Then, we can create groups of tasks each requiring no
more than total bandwidth of a disk. Thus, we can “line” them one after another and finish in k
units of time. Let us assume that we can schedule tasks to finish in k units of time where k is the
minimum such time. We can map schedule of every time-unit into contents of one bin. Thus, k
bins suffice.

**Remark 3.5:** Note that ARS is NP-hard in the strong sense since so is bin packing.

### 3.1.4 Heuristics

Since ARS is NP-hard, efficient heuristics should be developed. Furthermore, since in real-world
applications (such as video-on-demand) a scheduler does not have the complete knowledge about
all the tasks in advance, an on-line scheduler is desirable.

Another requirement by the real-world applications is that they do not care much about the
finishing time of a set of tasks. Their main objective is for the tasks to observe a minimum average
startup delay. The startup delay of a task t is defined as \( \sigma(t) - r(t) \). Hence, the objective of ARS
can be modified to be minimizing \( \sum_{t \in T} \sigma(t) - r(t) \). Note that the new objective does not impact
the interactivity of the ARS problem.

Showing a scheduling problem is NP-hard is not sufficient to justify an investigation for a good
heuristic. It is possible that a naive heuristic, no matter how bad it performs, always provides a
near-optimal solution. To show that this is not the case for ARS, consider the following example.

**Example 3.6:** Suppose \( D = 3 \) and \( n = 7 \). Given the task information as:
\[ t_i : < r(t_i), l(t_i), c(t_i), p(t_i) >, \]
then:
\[
\begin{align*}
t_1 : & < 0, 5, 1.0, 1 > & t_2 : & < 0, 5, 1.0, 2 > & t_3 : & < 0, 5, 1.0, 3 > \\
t_4 : & < 1, 100, 1.0, 1 > & t_5 : & < 2, 100, 1.0, 2 > & t_6 : & < 3, 100, 1.0, 3 > \\
t_7 : & < 4, 5, 1.0, 1 >
\end{align*}
\]

\(^4\)The precise equation is \( \sigma(t) - r(t) + 1 \) because of the cycle-based approach where the display is always one interval
behind the retrieval.
Assuming the FCFS scheduling policy, $t_1$, $t_2$, and $t_3$ utilize the entire system bandwidth from interval 0 to interval 4. At interval $u = 5$ suddenly the entire system bandwidth becomes available and three out of the remaining four tasks ($t_4$, $t_5$, $t_6$, $t_7$) can be initiated. FCFS selects tasks $t_4$, $t_5$, and $t_6$, based on their release time. Subsequently, $t_7$ cannot be initiated sooner than interval 105, resulting on an average latency time of $\frac{110}{7}$ intervals for this set of tasks. An alternative scheduling algorithm which selects tasks $t_5$, $t_6$, and $t_7$ to be initiated at interval $u = 5$, will observe an average latency time of $\frac{40}{7}$ intervals. Hence, this algorithm performs approximately 7 times better than FCFS.

The main structure for all the proposed heuristics to solve the ARS problem is as follows. The heuristics start with time interval $u = 0$ and increment $u$ until all $t \in T$ are scheduled. For each value of $u$ the following steps are taken respectively:

1. The required resources are assigned to the tasks that have already been started.
2. A set $T_x$ of un-scheduled tasks ($T_x \subseteq T$) is constructed of tasks that can start at $u$ without violating any constraint, i.e., $\forall t_x \in T_x: r(t_x) \leq u$ and $\forall i \leq D, \sum_{t \in (S(u) \cup \{t_x\})} R_i(t) \leq 1.0$, where $S(u)$ and $R_i(t)$ are as defined in Def. 3.1. In particular, $T_x$ is the set of the released tasks that do not result in retrieval contention if scheduled at $u$.
3. The set $T_x$ is sorted in a non-decreasing order, based on the value of $h(t)$ (the function $h(t)$ varies from one heuristic to the other).
4. The tasks are scheduled at $u$ consecutively from the beginning of the sorted set to its end. Before scheduling a task, it is examined that if any constraint will be violated assuming the tasks is scheduled. If so then the task will not be scheduled. Later we show that this examination is fast.

Note that all the tasks in $T_x$ usually cannot be simultaneously initiated at $u$ without violating any constraint. Hence, the question is: which subset of $T_x$ should be selected to be initiated at $u$? This is the phase that alternative heuristics perform differently by employing various $h(t)$ functions to sort and select from $T_x$. The reason that the proposed structure results in on-line scheduling algorithms is that at each time interval $u$, $T_x$ is constructed from the tasks that has already been arrived, i.e., $\forall t_x \in T_x: r(t_x) \leq u$. Therefore, the heuristic functions, $h(t)$, have all the required information about the tasks of $T_x$ in order to sort them. There are some famous heuristics discussed in the literature employed by alternative scheduling problems. The following is a list of some of those heuristics that are adopted by ARS with their corresponding $h$-function:
- **First Come First Serve (FCFS):** $h(t) = r(t)$.
- **First Fit Decreasing (FFD):** $h(t) = \frac{1}{c(t)}$.
- **Earliest Completion time First (ECF):** $h(t) = \ell(t)$.
- **Earliest Deadline First (EDF):** $h(t) = r(t) + \ell(t)$.
- **FFD + ECF:** $h_1(t) = \frac{1}{c(t)}$, $h_2(t) = \ell(t)$.

With the first four heuristics (FCFS, FFD, ECF, EDF), in case of a tie an arbitrary choice is made. With FFD + ECF first $h_1$ is employed, and in case of a tie, $h_2$ is employed as a tie breaker.

FCFS is the simplest heuristic which is described mostly for multimedia task scheduling [TPBG93, BGMJ94, DSS94]. Its rationale is that the sooner a task arrives the sooner it should be scheduled.

The FFD heuristic is based on a paper by Garey et. al. [GGJY76]. FFD was originally devised for the bin-packing [GJ79] problem. In the bin-packing problem, a finite number of variable-size items should be placed in a single unit-capacity “bin”, with the objective being to pack all the items in as few such bins as possible. To adopt it to ARS, each interval can be assumed as a bin and each task as an item. Subsequently, the objective is to schedule all the tasks in order to minimize the finishing time\(^5\) (as few time intervals as possible). The FFD heuristic fills a bin starting from the larger items that can fit in the bin, and hence its adopted version for ARS should start scheduling the tasks with a higher bandwidth requirement. The justification is that if there is some available bandwidth, then it should be dedicated to a task that can utilize it more because those requiring less bandwidth have a higher chance to be scheduled in the future.

To justify the ECF heuristic, consider the following discussion. Suppose that at time interval $i$ there are only enough resources available to start the retrieval of either $t_1$ or $t_2$, but not both. If we start $t_1$ and postpone $t_2$, assuming no new resources become available, $t_2$ will observe a startup delay of $\text{delay}(t_2) = (i + \ell(t_1)) - r(t_2)$ while $\text{delay}(t_1) = i - r(t_1)$. On the other hand, if $t_2$ is scheduled first then $\text{delay}(t_1) = (i + \ell(t_2)) - r(t_1)$ and $\text{delay}(t_2) = i - r(t_2)$. If we compute $\text{delay}(t_1) + \text{delay}(t_2)$ for each case and compare them with each other, we observe that the minimum

---

\(^5\)Note that although bin-packing and ARS have some similarities, they also have some major differences. For example, in ARS, scheduling a task $t$ will occupy $\ell(t)$ future bins. Furthermore, while a bin has enough space for an item (task), it might not be feasible to fit the task in the bin (time interval) because of the constrained placement.
among the two depends on the length of each task. In other words, if \( \ell(t_1) < \ell(t_2) \), then it is better to schedule \( t_1 \) and then \( t_2 \).

EDF is the most popular heuristic for real-time scheduling problems and is originally described in [Der74]. It is fine tuned for multiprocessor task scheduling by assigning deadlines to tasks such that the execution of each task should be completed prior to its deadline. In the case of ARS, suppose every tasks in \( T \) can tolerate a maximum startup delay of \( MaxDel \). Hence, the deadline for a task \( t \) can be computed as: \( r(t) + \ell(t) + MaxDel \). Therefore, among a set of tasks, the one with the minimum \( r(t) + \ell(t) \) has also the earliest deadline.

Finally, FFD+ECF is a combined heuristic that attempts to improve the performance of FFD in the event of a tie. It combines two heuristics that employ different parameters in their \( h \)-functions\(^6\). Simulations have shown (see Section 4) that FFD+ECF is superior to the others. However, when we eliminate the constraint placement by assuming a single disk whose bandwidth is identical to the aggregate bandwidth of multiple disks, ECF outperformed the rest. This implies that FFD+ECF captures the constraint placement characteristic of ARS, and supports the fact that ARS is a new scheduling problem that requires new attention. In particular, when \( D > 1 \) with a heavy system load, the probability that a task \( t \) finds the disk \( p(t) \) with sufficient bandwidth is lower than for when \( D = 1 \). With \( D = 1 \), as soon as a task \( t_1 \) commits and free \( c(t_1) \) of disk bandwidth, another task \( t_2 \) where \( c(t_2) \leq c(t_1) \) can start. However, with \( D > 1 \), another restriction should also be satisfied and that is the disk that becomes free should be the same as the disk containing the first subobject of \( t_2 \) (i.e., \( p(t_2) \)). For a heavy load system there is a high chance that the freed bandwidth become utilized by another task resulting in a higher startup delay for \( t_2 \). Since a task with a higher bandwidth requirement has a less chance to find sufficient disk bandwidth, it should be utilized as soon as the two restrictions are satisfied. This explains why FFD+ECF performs better than ECF when \( D > 1 \) while the reverse becomes true with \( D = 1 \).

Since the subobject assignment follows a regular pattern (round-robin), and the scheduling algorithms do not look ahead and reserve resources in advance, the proposed system is fast (see Section 2.2.1). To implement the above heuristics, a scheduler should maintain a queue of the

\(^6\)We also investigated other combinations such as ECF+FFD and EDF+FFD. However, we eliminated them from the paper because they did not result in any new observations besides those that are reported for other heuristics.
retrieval tasks. As soon as a request arrives a retrieval task is generated and added to the queue in the correct location. The correct location of the task in the queue is determined by the \( h \) function of the chosen heuristic (i.e., the queue should remain sorted based on \( h(t) \) after insertion). Note that no re-sorting is required, instead an insertion will be sufficient \( (O(\log n) \text{ vs. } O(n \log n)) \), where \( n \) is the maximum length of the queue. The scheduler maintains a data structure that determines the bandwidth consumption of each disk per interval. Assume the data structure is a matrix \( busy \) where each row corresponds to a time interval and each column to a disk. For example, \( busy(u, d) = 0.7 \) determines that 70% of the bandwidth of disk \( d \) is busy at time interval \( u \). At the beginning of each time interval, the scheduler starts from the head of the queue and decides if a task can be scheduled or not. Note that this decision making is fast because only the current interval and the required disk drive is examined for retrieval contention. In other words, to examine if task \( t \) can be scheduled at \( u \), it is sufficient to check the condition: \( busy(u, p(t)) + c(t) \leq 1.0 \). If the task can be initiated, it is removed from the queue and started. At this point, it is only required to update \( busy(u, p(t)) \). This procedure continues until the tail of the queue is reached. Now that the tasks are initiated and the scheduler has nothing else to do for the rest of the current time interval, it can update \( busy \) for the maximum length of the tasks that are scheduled. The first portion of the scheduler which should be invoked at the beginning of each time interval is fast and ideally can be implemented by the hardware. The portion which is in charge of adding tasks to the queue can also be implemented by the hardware. However, it is not a critical portion because in the worst case a task will observe a small extra startup delay because of its late insertion to the queue.

3.2 Augmented ARS (\( ARS^+ \))

We use the augmented ARS problem, termed \( ARS^+ \), as a transient phase between ARS and CRS. \( ARS^+ \) is identical to ARS except that each task \( t \) has a lag parameter, \( \delta(t) \), that determines the start time of the task. That is, the display of a task that is released at \( r(t) \) should not start sooner than \( r(t) + \delta(t) \). To motivate \( ARS^+ \), consider a video-on-demand application where the customers reserve movies in advance. For example, at 7:00 pm Alice reserves \emph{God Father} to be displayed at

\footnote{In real implementation it can be a link list where as the scheduling proceeds, the nodes prior to the current time interval are released.}
8:00 pm. Hence, assuming \( t \) be the task corresponding to Alice retrieving *Godfather*, \( r(t) = 7:00 \) and \( \hat{t}(t) = 1:00 \).

Suppose \( t \) is started at time interval \( u \), there are three possible scenarios: 1) \( r(t) \leq u < r(t) + \hat{t}(t) \), 2) \( u = r(t) + \hat{t}(t) \), or 3) \( u > r(t) + \hat{t}(t) \). Trivially, with the second and third scenarios \( t \) will observe a startup delay of zero and \( r(t) + \hat{t}(t) - u \) intervals, respectively. These two scenarios are similar to ARS where a task is started at or after its release time. The interesting case is the first scenario. In this case, although the retrieval of the corresponding object is started sooner than \( r(t) + \hat{t}(t) \), the display can still be initiated at \( r(t) + \hat{t}(t) \). This can be achieved by buffering the data retrieved prior to \( r(t) + \hat{t}(t) \), termed *upsliding* described\(^8\) in [CGS95]. Upsliding starts a retrieval task sooner than the display of its corresponding object. Memory buffers are employed to store those subobjects that are retrieved sooner and have not been displayed as yet. As the delay between retrieval and display increases, the amount of required memory also increases. To illustrate the concept of upsliding and its memory requirement consider the following example.

**Example 3.7**: Suppose: \( D = 3, T = \{t_1, t_2, t_3 \} \). Given the task information as:

\[
t_i : < r(t_i), \hat{t}(t_i), \ell(t_i), c(t_i), p(t_i) >,
\]

then:

\[
t_1 : < 0, 2, 5, 0.6, 1 > \quad t_2 : < 1, 1, 5, 0.6, 1 > \quad t_3 : < 2, 0, 5, 0.6, 1 >
\]

The required display schedules of these tasks as a function of time is depicted in Fig. 5b. The placement of objects \( X, Y \), and \( Z \) referenced respectively by \( t_1, t_2 \), and \( t_3 \) is shown in Fig. 6. Note that to simplify the example, all the tasks have equal start time (i.e., \( r(t) + \hat{t}(t) \)), length, display bandwidth, and their first subobjects reside on the same disk. Hence, if a scheduler starts all the tasks at their start time, there would be a retrieval contention for disk 1. One solution is to postpone the retrieval of \( t_2 \) and \( t_3 \) for one and two intervals, respectively, observing a total startup delay of 3 intervals for this collection of tasks. However, since the information about \( t_2 \) \((t_1)\) is released one (two) interval(s) sooner than its start time, its retrieval can start earlier by employing upsliding. In this case, no startup delay will be observed. Table 1 demonstrates the status of retrieval, memory, and display per time interval. ■

---

\(^8\)The difference is that in [CGS95], we introduced upsliding for binary single-media composite objects.
From the above example, the memory requirement for an upslided task has a growing phase, steady phase, and a shrinking phase. The detailed computation of memory requirement for each phase can be found in [SG95]. The important phase is the steady phase which determines the maximum memory requirement. The maximum memory requirement for each upslided task is proportional to both the number of upslided intervals and its display bandwidth requirements:

$$MaxMem(t) = \begin{cases} \frac{c(t) \times (r(t) + \frac{\tilde{a}(t)}{t} - \sigma(t))}{\frac{\tilde{a}(t)}{t} + \frac{\tilde{a}(t)}{t} - \sigma(t)} & \text{if } r(t) + \frac{\tilde{a}(t)}{t} > \sigma(t) \\ 0 & \text{otherwise} \end{cases}$$

Note that the amount of required memory in megabit can be computed as $MaxMem(t) \times RD \times interval$, where $interval$ is the duration of a time interval in seconds. This is because the bandwidth requirement of $t$ is $c(t) \times RD$ and the display time of one subobject of the object referenced by $t$ is $interval$ seconds. Since Equation 2 has already incorporated $c(t)$, it is sufficient to multiply $MaxMem(t)$ by $RD \times interval$ to compute the memory requirement in megabits.

**Theorem 3.8:** $ARS^+$ is $NP$-hard.

**Proof:** $ARS^+$ can be restricted to ARS by assuming $\forall t \in T \frac{\tilde{a}(t)}{t} = 0$. □

To describe the heuristics for $ARS^+$, assume the maximum amount of sliding for a task $t$ is denoted by $B(t)$. The main structure of the heuristics for $ARS^+$ is very similar to that of ARS (see Section 3.1.4). The difference is that at each interval $u$, the set $T_u$ is constructed from the tasks $t_x$ such that $\forall t_x \in T_u$:

\[\text{Table 1: System status with upsliding for Example 3.7}\]

<table>
<thead>
<tr>
<th>Time Interval</th>
<th>Retrieve</th>
<th>Memory</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>$X_2, Y_1$</td>
<td>$X_1, X_2, Y_1$</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$X_3, Y_2, Z_1$</td>
<td>$X_2, X_3, Y_2, X_1, Y_1, Z_1$</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>$X_4, Y_3, Z_2$</td>
<td>$X_3, X_4, Y_3, X_2, Y_2, Z_2, X_1, Y_1, Z_1$</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>$X_5, Y_4, Z_3$</td>
<td>$X_4, X_5, Y_4, X_3, Y_3, Z_3, X_2, Y_2, Z_2$</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>$Y_5, Z_4$</td>
<td>$X_5, Y_5$</td>
<td>$X_4, Y_4, Z_4, X_3, Y_3, Z_3$</td>
</tr>
<tr>
<td>6</td>
<td>$Z_5$</td>
<td>-</td>
<td>$X_5, Y_5, Z_5$</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>$X_5, Y_5, Z_5$</td>
</tr>
</tbody>
</table>

\[\text{Note that the amount of required memory in megabit can be computed as } MaxMem(t) \times RD \times interval, \text{where } interval \text{ is the duration of a time interval in seconds. This is because the bandwidth requirement of } t \text{ is } c(t) \times RD \text{ and the display time of one subobject of the object referenced by } t \text{ is } interval \text{ seconds. Since Equation 2 has already incorporated } c(t), \text{ it is sufficient to multiply } MaxMem(t) \text{ by } RD \times interval \text{ to compute the memory requirement in megabits.}\]

\[\text{Theorem 3.8: } ARS^+ \text{ is } NP\text{-hard.}\]

\[\text{Proof: } ARS^+ \text{ can be restricted to ARS by assuming } \forall t \in T \frac{\tilde{a}(t)}{t} = 0. \]

\[\text{To describe the heuristics for } ARS^+, \text{ assume the maximum amount of sliding for a task } t \text{ is denoted by } B(t). \text{ The main structure of the heuristics for } ARS^+ \text{ is very similar to that of ARS (see Section 3.1.4). The difference is that at each interval } u, \text{ the set } T_u \text{ is constructed from the tasks } t_x \text{ such that } \forall t_x \in T_u:}\]

\[\text{In [SG95], we assumed a single media type. The extension of equations to a mix of media type can be done similar to Eq. 2 which computes the memory requirement for the steady phase.}\]
1. $u \geq r(t_x)$

2. $r(t_x) + \beta(t_x) - u \leq B(t_x)$

3. $t_x$ can be scheduled at $u$ with no retrieval contention, i.e., $\forall i \leq D, \sum_{t \in (S(u) \cup \{t_x\})} R_i(t) \leq 1.0$, where $S(u)$ and $R_i(t)$ are as defined in Def. 3.1.

4. $t_x$ can be scheduled at $u$ without exhausting the system memory (SYSMEM), i.e., assuming $\sigma(t_x) \leftarrow u$ then $\sum_{t \in (S(u) \cup \{t_x\})} MaxMem(t) \leq SYSMEM$.

Previously, with ARS, as soon as a task was released (item 1) and it resulted in no retrieval contention (item 3), it was added to $T_x$. Here, however, a task can start before its start time (item 2) if it is allowed to (based on $B(t)$) and it will be added to $T_x$ if besides all the mentioned constraints, the system memory does not overflow (item 4). Once $T_x$ is constructed, alternative methods to sort $T_x$ and the procedure of selecting tasks will be exactly identical to those of ARS heuristics. Indeed, the same heuristics and $h$ functions can be employed.

The values of $B(t)$ play an important role in all the heuristics. It determines how soon the scheduler can start a task $t$, although due to retrieval or memory contention it might not be able to initiate $t$. Indeed, $B(t)$ determines the share of task $t$ from the system memory. If $\forall t \in T B(t) = 0$, then no sliding is allowed. In this case, the ARS and $ARS^+$ heuristics are identical because among the above four restrictions to add a task to $T_x$, item 2 is redundant as it is identical to item 1 (assuming $r(t) \leftarrow r(t) + \beta(t)$) and item 4 is always true ($\forall t \in T MaxMem(t) = 0$). In Section 4, the optimal value of $B$ is both computed analytically and verified using a simulation study.

Since the main structure of the $ARS^+$ scheduling algorithm is identical to that of the ARS, the same implementation techniques described for ARS (see Section 3.1.4) can be applied here. The sole reason that the main structure of the $ARS^+$ heuristics are constructed similar to that of the ARS, is to maintain the speed of the scheduling algorithms. Otherwise, a scheduler that attempts to schedule a task $t$ at $r(t) + \beta(t)$ and in case of failure slides $t$ upward, might be more successful. However, the problem with this algorithm is that it needs to look ahead and reserve resources in advance, resulting in slow scheduling (see the discussion of Section 2.2.1). To observe, assume $r(t) = 5$, and $\beta(t) = 4$, i.e., $t$ should start at interval 9. If at time interval $u$, where $u < 9$, the scheduler decides to schedule $t$ at 9, it should reserve resources for $t$ in advance. Hence, the only time that the scheduler can initiate $t$ at time 9 without advance resource reservation is at
Howev er, time interval $u = 9$ is too late to decide for upsliding a task because the system has already missed the previous intervals. This explains why to schedule a task $t$, our proposed algorithm starts from $r(t) + \delta(t) - B(t)$ and proceeds down the time intervals, instead of starting from $r(t) + \delta(t)$ and analyzing the previous time intervals.

### 3.3 Composite Retrieval Scheduling (CRS)

With CRS, each composite task consists of a number of atomic tasks. We use $t$ to represent an atomic task and $\theta$ for a composite task. Similarly, $T$ represent a set of atomic tasks while $\Theta$ is a set of composite tasks. A composite task, itself, is a set of atomic tasks, e.g., $\theta = \{t_1, t_2, ..., t_n\}$. Each atomic task has the same parameters as defined in Section 3.1.1, except for the release time $r(t)$. Instead, each atomic task has a lag time denoted by $\delta(t)$. Without loss of generality, we assume for a composite task $\theta$, $\delta(t_1) \leq \delta(t_2) \leq ... \leq \delta(t_n)$. Subsequently, we denote the first atomic task in the set $\theta$ as $f(\theta)$, i.e., $f(\theta) = t_1$. Lag time of a task is identical to the lag parameter (see Section 2.3) of its corresponding object and determines the start time of the task with respect to $\delta(f(\theta))$. Trivially, $\delta(f(\theta)) = 0$. Each composite task, on the other hand, has only a release time $r(\theta)$ which is the time that a request for the corresponding composite object is submitted.

**Definition 3.9:** An atomic task $t$ is schedulable at $u$ iff $t$ can be started at $u$ and completes at $u + \ell(t) - 1$ without violating any resource constraint as defined in Def. 3.1. \[\square\]

**Definition 3.10:** A composite task $\theta = \{t_1, t_2, ..., t_n\}$ is said to be schedulable at $u$ (i.e., $\sigma(\theta) \leftarrow u$) iff $\forall t \in \theta$, $t$ is schedulable at $u + \delta(t)$ (i.e., $\sigma(t) \leftarrow u + \delta(t)$). \[\square\]

Based on the above definition, the CRS problem can be defined similar to the ARS problem. The definition of CRS along with its proof of $NP$-hardness is provided in Appendix A.

### 3.3.1 Approaches to Solve CRS

There are two general approaches to solve the CRS problem. One is to view a composite task as a set of independent atomic tasks. The other is to view a composite task as a single atomic task.
As is explained below, either of these approaches increases complexity in scheduling.

The first approach constructs a set $T$ as $T = \bigcup_{\theta \in \Theta} \theta$. The release time for an atomic task $t \in \theta$ can be computed as: $r(t) = r(\theta)$, i.e., an atomic task is released once its corresponding composite task is released. Now the CRS problem can be viewed as the $ARS^+$ problem that should schedule the set $T$. The main difference is that for those $t$ that are not the first task of any composite task (i.e., $\exists \theta \in \Theta$ s.t. $t = f(\theta)$), they should either be scheduled immediately at their start time (i.e., $r(t) + \delta(t)$) or the display of their corresponding composite object will suffer from hiccups. In other words, they cannot tolerate any startup delay once the display of their composite task has started. The only case that $t \in \theta$ can tolerate a startup delay of $\Delta$ is when $f(\theta)$ observes the startup delay of $\Delta$. Adding this constraint to the $ARS^+$ problem increases the complexity of scheduler because it is now forced to schedule tasks independently while they depend on each other due to the structure of the composition.

An alternative approach is to schedule composite tasks as if they are atomic tasks, employing $ARS$ heuristics. To achieve this, we should assign a length ($\ell$) and a bandwidth requirement ($c(t)$) to each composite task so that the heuristics be able to prefer one task to the other. Towards that end, for each composite task $\theta$ with $n$ atomic tasks we define:

$$c(\theta) = \sum_{t \in \theta} \frac{c(t)}{n}$$

$$\ell(\theta) = \max_{t \in \theta}(\delta(t) + \ell(t))$$

If more than one composite task is schedulable at $u$, then one of the $ARS$ heuristics is employed to select one (or more) task(s). The definition of a composite task being schedulable at $u$ is as defined in Def. 3.10. However, we should emphasize that the use of $c(\theta)$ and $\ell(\theta)$ is to only enable $ARS$ heuristics to compute $h(\theta)$. As in the previous approach, the key problem with this approach is also to verify schedulability, i.e., it is more complex to infer retrieval contention.

To provide an analogy on how the first and second approaches are different, consider Fig. 7. In Fig. 7a each shaded box is an atomic task and the problem of $ARS$ is to fit them all in the big rectangle on the right with the objective to minimize the amount of required time by consuming as much of the available disk bandwidth as possible. With CRS the problem of filling the rectangle is identical to $ARS$, however, each task is now a polygon (instead of a rectangle). The first approach
strives to partition a polygon into boxes so that it can use the $ARS^+$ solutions to solve CRS.

The problem is that the $ARS^+$ solutions might destroy the shape of the polygons in the process of scheduling. Hence, more constraints should be added to $ARS^+$. The second approach, on the other hand, maintains the shape of the polygons while scheduling them. Instead, it employs ARS solutions to select a polygon among all the polygons that can fit from a certain point. In this paper we focus on the second approach and do not consider the first further.

We now return to the problem of determining retrieval contention with the second approach. The key point is that the structure of a composite task does not follow a regular pattern, thus violating the first criteria for fast scheduling (see Section 2.2.1). Hence, to schedule a composite task $\theta$ at $u$, the entire duration from $u$ to $u + \ell(\theta)$ should be examined for retrieval contention. In other words, advance resource reservation via look-ahead becomes essential. Implementation of this technique demands choosing longer time interval in order to compensate for the extended duration of computation of the scheduler. This results in less number of simultaneous displays per interval compared to a system with the same system load which does not support composite objects.

### 3.3.2 The Critical Role of Memory

As we have shown in Section 3.2, scheduling of $ARS^+$ can benefit from memory. However, in case of CRS, memory may be necessary to achieve schedulability, as discussed below.

A composite object may have internal contention, i.e., atomic tasks that constitute a composite task may compete with one another for the available disk bandwidth. Hence, it is possible that
due to \textit{internal contention}, a composite task cannot be initiated at any time interval independent of the system load. To observe, consider a composite task \( \theta \) that consists of the atomic tasks of Example 3.2. Note that the release time of the tasks of this example should be considered as their lag time. Independent of the time interval that \( \theta \) is initiated, there is an internal contention between \( t_1 \) and \( t_3 \) after 4 intervals from the start time of \( \theta \). In other words, it is not possible to start all the atomic tasks of \( \theta \) at their start time.

\textbf{Definition 3.11: Internal contention:} Consider a composite task \( \theta = \{t_1, t_2, ..., t_n\} \) and \( \forall u \geq 0 \) let \( S(u) \subseteq \theta \) be the set of atomic tasks which \( \frac{2}{3}(t) \leq u < \frac{2}{3}(t) + \ell(t) \). The composite task \( \theta \) has internal contention iff \( \exists u, 1 \leq i \leq D \) such that \( \sum_{t \in S(u)} R_i(t) > 1.0 \) where:

\[
R_i(t) = \begin{cases} 
  c(t) & \text{if } (p(t) + u - \frac{2}{3}(t)) \mod D = i - 1 \\
  0.0 & \text{otherwise}
\end{cases}
\]

The above definition, intuitively means that the retrieval of all the atomic tasks of \( \theta \) cannot be scheduled at their start time without violating the bandwidth constraint. In other words, \( \not\exists u \) such that \( \theta \) be schedulable at \( u \) (see Def. 3.10). Using our graphical analogy, the width of the polygon that represents a composite task with internal contention is larger than the width of the rectangle (see Fig. 7).

To resolve internal contention, one should move atomic tasks of \( \theta \) so that the deformed polygon can fit in the rectangle. This can be achieved by \textit{buffered sliding} described in [CGS95]. Resolving the internal contention for a composite task \( \theta \) is to modify the start time of its consisting atomic tasks, such that Def. 3.11 does not hold true for \( \theta \). Such a modification requires use of buffers\(^{10}\). Ideally, we should minimize the amount of required buffer. There are two types of buffered sliding for composite tasks: \textit{upsliding} and \textit{downsliding}. Consider each case in turn.

Upsliding is similar to the discussion of Section 3.2: Its resolving of internal contention for a composite task \( \theta \) is the same as solving \( \text{ARS}^+ \) for a set of atomic tasks \( \theta \) where \( \forall t \in \theta \ r(t) = r(\theta) \). Once this problem is solved, the start time of atomic tasks can be updated to their scheduled time.

\(^{10}\)We use the terms buffer and memory interchangeably in this section.
(i.e., \( \delta(t) \leftarrow \sigma(t) - r(t) \)) and the modified composite task will have no internal contention. Since resolving internal contention is equivalent to solving \( ARS^+ \),

**Theorem 3.12:** Resolving internal contention for a composite task by upsliding is \( NP \)-hard.

Employing \( ARS^+ \), it is possible that a task starts after its start time. This can also be compensated for by memory as described later as part of a downsliding approach. Note that the objective of \( ARS^+ \) should be modified to minimize the amount of required memory (instead of minimizing the average startup delay). This is achieved by minimizing \( \sum_{t \in T} |\sigma(t) - r(t) - \delta(t)| \) because the difference between the scheduled time of a task and its start time determines the amount of required memory.

An alternative approach to resolving internal contention is downsliding. With this approach, the start time of constituting atomic tasks of a composite task is postponed (moved downward). To compensate for this delay, the display of the entire composite object is delayed. To illustrate, consider the following example.

**Example 3.13:** Suppose: \( D = 3, \theta = \{t_1, t_2, t_3\}, \) and \( r(\theta) = 0 \). Given the task information as:

\[ t_i :< \delta(t_i), \ell(t_i), c(t_i), p(t_i) >, \]  then:

\[ t_1 :< 0, 5, 0, 8 >, t_2 :< 2, 5, 0, 6 >, t_3 :< 2, 5, 0, 6 >. \]

The required display schedules of these tasks as a function of time is depicted in Fig. 5c. The placement of objects \( V, Y, \) and \( Z \) referenced respectively by \( t_1, t_2, \) and \( t_3 \) is shown in Fig. 6. If all the tasks is initiated at their start time (i.e., \( r(\theta) + \delta(t) \)), there will be a retrieval contention for disk 1 at \( u = 2 \). However, the system can downslide the retrieval of \( t_2, t_3 \) by one (two) interval(s) while the retrieval of \( \theta \) still starts at 0. In this case, the display of \( \theta \) will observes a startup delay of two intervals. Table 2 demonstrates the status of retrieval, memory, and display per time interval.

The amount of startup delay introduced by downsliding for composite task \( \theta \) is:

\[ delay(\theta) = \text{Max}_{t \in \theta}(\sigma(t) - r(t) - \delta(t)) \]  \hspace{1cm} (6)
The memory requirement of a composite task contributed by downsiding for each time interval $u$ is:

$$\text{Mem}(u) = \text{Mem}(u - 1) + \text{Ret}(u) - \text{Disp}(u)$$

where, $\text{Ret}(u)$ and $\text{Disp}(u)$ are the amount of data retrieved and displayed at $u$, respectively. That is, whatever remains should be accumulated in memory. Subsequently, $\text{Ret}(u)$ and $\text{Disp}(u)$ are computed as follows.

$$\text{Disp}(u) = \sum_{t \in \theta} \text{Disp}(t, u)$$

$$\text{Disp}(t, u) = \begin{cases} 
  c(t) & \text{if } r(t) + \frac{\theta(t)}{2} + \text{delay}(\theta) \leq u < r(t) + \frac{\theta(t)}{2} + \text{delay}(\theta) + \ell(t) \\
  0 & \text{otherwise}
\end{cases}$$

$$\text{Ret}(u) = \sum_{t \in \theta} \text{Ret}(t, u)$$

$$\text{Ret}(t, u) = \begin{cases} 
  c(t) & \text{if } \sigma(t) \leq u < \sigma(t) + \ell(t) \\
  0 & \text{otherwise}
\end{cases}$$

Assuming only downsiding, resolving internal contention becomes very similar to solving the ARS problem. A composite task can be considered as $T$ in ARS, and the start time of the tasks as their release time. Subsequently, minimizing the startup delay can be translated directly to minimizing the memory requirement. Hence,

**Theorem 3.14:** Resolving internal contention by downsiding is NP-hard.

Due to the above argument, heuristics for solving ARS problem can be employed to resolve the internal contention as well. That is, as a pre-processing stage, one can apply the heuristics on...
those composite tasks that contain internal contention in order to modify the start time of their corresponding atomic tasks. The resulting set of composite tasks can be scheduled as described before.

4 Performance Evaluation

In this section we report the results of our experiments on $ARS^+$. We focused on $ARS^+$ for two reasons. First, ARS is a special case of $ARS^+$. Second, CRS can be solved by extending the solutions of ARS and $ARS^+$ (as described in Section 3.3). Hence, the observations with $ARS^+$ hold true for both ARS and CRS.

We implemented a simulation model to: 1) compare the performance of alternative heuristics, and 2) investigate the impact of $B$ and system memory on their performances. First, we describe the simulation model. Next, we report the results of our experiments.

4.1 Simulation Model

For the purposes of this evaluation, we assumed a hardware platform consisting of 16 disk clusters (i.e., $D = 16$) each consisting of two Seagate ST31200W disks. Hence, the effective bandwidth of each cluster is 50 Mb/s. The amount of memory available for buffered sliding is\footnote{Note that for a system with 32 physical disk drives, this is a reasonable amount of memory, conceptually translated to 39 MB of memory per physical disk.} 1.22 Gigabyte (GB).

We assumed a news-on-demand application where objects belong to either of the four different media types shown in Table 3. The subobject sizes are computed assuming that the duration of a time interval is 2 seconds. The size of the clips ($l(t)$) is random and varies from 2 seconds (one subobject) up to 9.34 minutes (280 subobjects). The objects are assigned to the disks in a round-robin manner, starting with a random disk.

We employed an open simulation model for our evaluation: the number of tasks ($n = 4000$) released every minute is controlled by a Poisson distribution. We manipulated the Poisson param-
<table>
<thead>
<tr>
<th>Media type</th>
<th>Consumption rate (Mb/s)</th>
<th>Subobject size (MB)</th>
<th>c(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG-1 (or stereo CD quality audio)</td>
<td>1.5</td>
<td>0.38</td>
<td>0.03</td>
</tr>
<tr>
<td>Low quality MPEG-2</td>
<td>5</td>
<td>1.25</td>
<td>0.1</td>
</tr>
<tr>
<td>High quality MPEG-2</td>
<td>15</td>
<td>3.75</td>
<td>0.3</td>
</tr>
<tr>
<td>Compressed HDTV</td>
<td>35</td>
<td>8.75</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 3: Media parameters

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>Light</th>
<th>Moderate</th>
<th>Heavy</th>
<th>Imaginary disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>1.74</td>
<td>24.28</td>
<td>592</td>
<td>0</td>
</tr>
<tr>
<td>FFD</td>
<td>1.75</td>
<td>28.92</td>
<td>563</td>
<td>0</td>
</tr>
<tr>
<td>ECF</td>
<td>1.75</td>
<td>22.45</td>
<td>441</td>
<td>0</td>
</tr>
<tr>
<td>EDF</td>
<td>1.80</td>
<td>26.87</td>
<td>578</td>
<td>0</td>
</tr>
<tr>
<td>FFD+ECF</td>
<td>1.69</td>
<td>21.27</td>
<td>379</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: Comparison of alternative heuristics

In the experiment, we tested four alternative system loads: Very light (6 tasks per minute), light (7.5 tasks per minute), moderate (10 tasks per minute), and heavy (15 tasks per minute) system load. The lag parameter, $\frac{D}{t}$, varies from 0 to 40 intervals (80 seconds).

4.2 Experimental Results

We report the results of two sets of experiments. In the first set, we compared the performance of different heuristics. For the sake of comparison we executed all the experiments not only on a platform of 16 clusters, but also on an imaginary disk whose bandwidth is identical to the aggregate bandwidth of those 16 clusters. Note that to construct such an imaginary disk in real world, more than 32 physical Seagate disks are required\textsuperscript{12}. In this imaginary disk the placement of data and round-robin scheduling of resources on behalf of a display becomes redundant, providing a data independent evaluation of the alternative proposed heuristics.

Table 4 presents the average startup delay (in number of intervals) observed with alternative heuristics. The difference between the average startup delay observed by each of FCFS, FFD, and EDF, with $D = 16$ and the imaginary disk is not significant. This is because at high system

\textsuperscript{12}This is because this platform is not scalable (see [GK95]).
Figure 8: Heavy (15 requests per minute)
load the disks become completely utilized with $D = 16$ due to the round-robin assignment of the subobjects, simulating a single disk with an aggregate bandwidth of 16 disks. However, for a lower system load (columns 2, 3, 5, and 6) 16 disk platform results in a higher average startup delay because the placement of data forces some tasks to wait until the disk containing their first subobject ($p(t)$) becomes available. In general, the performance of the heuristics becomes closer to each other as the system load decreases. The reason is that for low system load the queue generated at each interval does not have enough elements for a heuristic to make a difference. With a heavily loaded system, however, both FFD+ECF and ECF outperform other heuristics with $D = 16$ and imaginary disk, respectively. This shows that considering the bandwidth of the tasks when ordering the queue is beneficial to resolve contention on multiple disks while it is not crucial when all the tasks are competing for a single disk (see Section 3.1). To determine whether this is a general trend or a phenomena that occurs by chance due to the start times of tasks $(r(t) + \frac{S}{2}(t))$, we analyzed alternative scenarios by varying $\frac{S}{2}(t)$. Furthermore, since the same trend holds from light to heavy system load in Table 4, we selected the heavy system load because in this case the performance of the heuristics differs significantly (larger queues are constructed). The obtained results (see Fig. 8) demonstrates while FFD+ECF is superior to other heuristics with $D = 16$, ECF is superior with the imaginary disk. FFD+ECF outperformed (with $D = 16$) FCFS by 49.5% (on average).

The reduction in average startup delay is not for free. Sometimes one or more tasks might starve with both ECF and FFD+ECF. In the real implementation, a threshold can be determined for the maximum tolerable startup delay and once a task exceeds that threshold, it should be moved to the head of the queue.

In a second set of experiments, we investigated the impact of sliding on the average startup delay. We varied the value of $B(t)$ from 0 up to 20 and compared the average startup delay observed by both FCFS and FFD+ECF. A theoretically optimal value for $B(t)$ is the one that satisfies the following equation:

$$\sum_{t \in S(u)} B(t) \times Sub(t) = sysmem$$

(10)

where $S(u)$ denotes the set of tasks executed at $u$, $Sub(t)$ is the subobject size of the object.

---

13 We could not vary $r(t)$ because it results in a different load.
referenced by \( t \), and \( \text{sysmem} \) is the available system memory in MB. Given a task \( t_{\text{big}} \) where \( c(t_{\text{big}}) = 1 \), we can compute \( \text{Sub}(t_{\text{big}}) = \frac{R_D \times \text{interval}}{8} \) which is the size of its subobject in MB. Subsequently, for any \( t \), \( \text{Sub}(t) = \text{Sub}(t_{\text{big}}) \times \frac{c(t)}{c(t_{\text{big}})} \) or \( \text{Sub}(t) = \frac{R_D \times \text{interval} \times c(t)}{8} \). Substituting \( \text{Sub}(t) \) in Equation 10 and assuming a fix value \( (\text{Opt}B) \) for all \( B(t) \), we obtain:

\[
\frac{R_D \times \text{interval} \times \text{Opt}B}{8} \times \sum_{t \in S(u)} c(t) = \text{sysmem}
\]

Note that in the worst case \( \sum_{t \in S(u)} c(t) = D \), and hence from Equation 11 we obtain:

\[
\text{Opt}B = \frac{\text{sysmem} \times 8}{R_D \times \text{interval} \times D}
\]

The value of \( \text{Opt}B \) for our experiments is 6.25. In Fig. 9a, as we increase the value of \( B \), the average startup delay decreases until \( B = 10 \). Assigning more than 10 buffers to each task for sliding results in competition for the available memory, resulting in a non-deterministic performance of the system. The reason that this phenomena did not happen at \( B = 6.25 \) is because the system load is very low and \( \sum_{t \in S(u)} c(t) < D \). With a low system load (Fig. 9b), we observe that the system becomes non-deterministic when \( B \) increases beyond 6. With a moderate system load (Fig. 9c), this behavior occurs with smaller values of \( B \). The reduction in the average startup delay with buffered sliding ranges from 3\% to 950\% depending on the system load.

We also examined the value of \( B(t) \) to be a function of both \( \frac{s(t)}{B} \) (i.e., \( B(t) = \frac{s(t)}{B} \)) and \( \text{sysmem} \) (i.e., \( B(t) = \frac{\text{sysmem} \times 8}{R_D \times \text{interval} \times B} \)). In each case we varied the value of \( B \). With \( B(t) = \frac{s(t)}{B} \), the
observations were similar to those reported for a constant $B$. However, the reduction in the average startup delay was less than when $B$ is fixed. The second case (where $B(t) = \frac{\text{sysmem} \times 8}{R \times \text{interval} \times B}$) has already been covered because it is a subset of the reported cases. To illustrate, note that all the tasks are assigned an identical $B(t)$ value. Subsequently, by fixing $\text{sysmem}$ at 1.22 GB and varying $B$, different values generated for $B(t)$ are a subset of the reported values.

5 Related work

Several studies have investigated techniques to support the continuous display of atomic multimedia objects [Pol91, GS93, TPBG93, RV93, CL93, GHBC94, BGMJ94]. There are others that study the problem of scheduling the retrievals of the subobjects of different atomic objects during a time interval in order to utilize more of the disk bandwidth (and less memory) by eliminating the impact of the seek time [YCK92, BMC94, GHBC94, GKS95, CBR95, OBRS95]. However, we are only aware of few studies [DSS94, VGG94, TPBG93] that are related to scheduling the retrievals of atomic objects. Our work is different from all of the above in that our scheduling techniques rely on and exploit the regular pattern (round-robin) in which a task utilizes the disks. In addition, we focus on a heavily loaded system while in [TPBG93] it is assumed that the number of active requests is less than the number of streams supported by the system (light system load). In [VGG94], they focus on the methods to admit a submitted request without violating the quality of service for the active requests. However, the policy used to choose a task from a set of queued requests has not been addressed in [VGG94]. The contention prediction in our scheduling algorithms (see Section 2.2.1) is similar in principle to admission control policies. However, due to our focus on round-robin activation of the disks, the techniques adopted in our work are different. Finally, Dan et. al. [DSS94] propose a batching technique to group the requests that reference the same video in order to reduce the retrieval load. This technique can be adopted and tuned to our scheduling algorithms. In this case, each task can be viewed as a representative of a collection of requests referencing the same object.

When compared with the atomic objects, the retrieval of composite objects using a multi-disk hardware platform has received even less attention. A number of studies have analyzed composite
objects and temporal relationships from a conceptual perspective [Her90, VM93, LG93, HFK95]. Their focus is on alternative abstract representations of composite objects. Other studies [Ste90, LG90, RR93] concentrate on the networking aspects of a geographically distributed system and propose techniques to support the temporal relationships of a composite object when displayed at a remote client. A survey of these studies is presented in [Buf94, RR93]. In [CGS95], we proposed a technique (buffered sliding) to avoid retrieval contention when scheduling the retrieval of a single binary composite object (i.e., consisting of two atomic objects) belonging to a single media type. This study is a generalization of [CGS95] to support: 1) a mix of media types and 2) \(n\)-ary composite objects consisting of a mix of media types. Moreover, the approaches described in this study are more general and subsume those of [CGS95].

ARS might appear to be similar to the problem of real-time scheduling of tasks on multiprocessors (from [MC69, LL73] to [KS94]). One might conceptualize a disk as a processor to reduce ARS to a multiprocessor task scheduling problem. However, the constrained placement of data across the disks introduces new challenges. In multiprocessor task scheduling, a task can be scheduled for execution on any processor at any time, however, with ARS the retrieval of object \(X\) must start with the disk containing its first subobject and employ other disks in a round-robin manner. Stankovic et al. suggests that the placement constraints and the impact of this placement on the run time scheduling is an open research area [SSNB93, SSNB95].

An alternative reduction of ARS is to consider each disk as a resource. This suggests the similarity of ARS with the task scheduling under resource constraint [GJ75, GGJY76, Bla79, ZR87, WL95]. One major characteristic of ARS differentiates it from these studies. All these studies assume that the resources are occupied by a task during its life time. However, with ARS, a display acquires and releases resources periodically in a round-robin manner. A method to compensate for this is to break a task into subtasks, where each subtask acquires resources for its entire duration. Subsequently, a precedence relation [Law73] can be employed to relate the subtasks of a task. However, the precedence relation is not restricted enough for ARS. If a partial order is assumed such that task \(t_1\) precedes task \(t_2\), then this means that \(t_1\) should complete before \(t_2\) starts. However, with ARS, assuming \(t_1\) and \(t_2\) correspond to retrieval subtasks from disks \(i\) and \(i + 1\) on behalf of a single request, then \(t_2\) should start immediately after \(t_1\) completes. Otherwise the display may
suffer from hiccup. Hence, it is not sufficient to start $t_2$ any time after the completion of $t_1$.

6 Conclusions and Future Research Directions

We investigated a class of scheduling problems ($ARS$, $ARS^+$, CRS) covering a large number of multimedia applications. Due to the special requirements of these applications, they introduce new challenges to the conventional scheduling problems. Hence, we showed their uniqueness and concluded that the solutions proposed to the conventional scheduling problems need to be revisited and adopted for $ARS$, $ARS^+$, and CRS. We formalized these problems, proved their $NP$-hardness, and developed some heuristics (FCFS, FFD, ECF, EDF, FFD+ECF). These heuristics are appropriate for a heavy system load that results in the formation of a queue of pending tasks. Using a simulation model, we concluded that FFD+ECF is superior to the other alternatives. However, as the number of disks ($D$) decreases, ECF becomes more effective and eventually it outperforms $FFD + ECF$ when $D = 1$. This implies that FFD+ECF captures the constraint placement characteristic of $ARS$, and supports the fact that $ARS$ is a new scheduling problem that requires new attention. We also demonstrated that by employing buffers, the average startup delay can be decreased further when the system load is low. Note that buffer sliding is unavoidable (independent of the system load) to resolve internal contention for composite tasks. The optimal share of each task from memory used for buffered sliding was computed analytically and it was verified using a simulation study.

This study can be extended in two ways. First, the role of buffered sliding to resolve contention between atomic tasks of different composite tasks should be investigated. This will result in a more flexible and complex scheduling algorithms. This investigation can be done in the context of the first approach for solving CRS where atomic tasks are scheduled independently. In this case, adding a constraint to the atomic tasks such that their start time and release time become a function of the release time of the composite task is essential.

Second, in this study we focused on the demand driven paradigm [SG95]. That is, a task is released based on a submitted request. However, tasks might be released in a regular manner based on a data driven paradigm [SG95]. To illustrate such an environment, suppose that there is
a large database of digitized movies. A number of Cable companies (e.g., HBO, Showtimes, Movie Channel) might render this database to serve their customers. The schedules of Cable companies are known in advance. This can be the schedule of a day, a week, or a month. Meanwhile, a pay-per-view service provider might use the same database to display movies for its customers on demand. It is worthwhile to study the scheduling problems in a combined environment that incorporates both data and demand driven paradigms.

References


A CRS is \( NP \)-hard

**Definition A.1:** The problem of CRS is to find a schedule \( \sigma \) (where \( \sigma : \Theta \rightarrow N \)) for a set \( \Theta \), such that:

1) it minimizes the \textit{finishing time} \([GG75]w\), where \( w \) is the least time at which all tasks of \( \Theta \) have been completed, and 2) satisfies the following constraints:

- \( \forall \theta \in \Theta, \sigma(\theta) \geq r(\theta) \).
- \( \forall u \geq 0 \), let \( S(u) \) be the set of atomic tasks which \( \sigma(t) \leq u < \sigma(t) + \ell(t) \), then: \( \forall i, 1 \leq i \leq D \sum_{t \in S(u)} R_i(t) \leq 1.0 \) where:

\[
R_i(t) = \begin{cases} 
    c(t) & \text{if } (p(t) + u - \sigma(t)) \mod D = i - 1 \\
    0.0 & \text{otherwise}
\end{cases}
\]  

**Theorem A.2:** CRS is an \( NP \)-hard problem.

**Proof:** We restrict CRS to ARS by assuming all the composite tasks are singleton sets. Furthermore, for a composite task \( \theta = \{t\} \), we consider \( l(\theta) = \ell(t) \), \( p(\theta) = p(t) \), and \( r(\theta) = r(t) \). Hence, the restricted CRS becomes identical to ARS.