On Configuring Hierarchical Multimedia Storage Managers

Shahram Ghandeharizadeh, Hsun-Ko Chan, Martha L. Escobar-Molano and Xiangyu Ju

Computer Science Department
University of Southern California

February 3, 1994

Abstract

Multimedia information systems have emerged as an essential component of many application domains ranging from library information systems to entertainment technology. A challenging task when implementing these systems is to support a continuous display of multimedia objects. The challenging is due to the low I/O bandwidth of the current disk technology, the high bandwidth requirement of multimedia objects, and the large size of these objects which requires them to be almost always disk resident. One approach to resolve this limitation is to decluster a multimedia object across multiple disk drives in order to employ the aggregate bandwidth of several disks to support its continuous retrieval (and display). To provide on-line access to vast amount of data economically, the storage architecture of these systems is expected to be hierarchical.

Assuming a hierarchical storage manager that consists of some memory, D disk drives, and a tertiary storage device, this paper describes: 1) a technique to support a continuous display of possibly compressed multimedia objects, and 2) the fundamental factors that impact a choice of configuration parameters for the system, and an algorithm to compute them.

1 Introduction

During the past decade, information technology has evolved to store and retrieve multimedia data (e.g., audio, video). Multimedia information systems utilize a variety of human senses to provide effective means of conveying information. Already, these systems play a major role in educational applications, entertainment technology, and library information systems. A challenging task when implementing these systems is to support a continuous retrieval of an object at the bandwidth required by its media type [SAD*93, MWS93, GRAQ91] (in order to ensure its continuous display). This is challenging because certain media types, in particular video, require very high bandwidths. For example, the bandwidth required by NTSC\textsuperscript{1} for “network-quality” uncompressed video is approximately 45 megabits per second (mbps) [Has89]. Recommendation 601 of the International

\textsuperscript{1}The US standard established by the National Television System Committee.
Radio Consultative Committee (CCIR) calls for a 216 mbps bandwidth for video objects. A video object based on the HDTV (High Definition Television) quality images requires approximately a 836 mbps bandwidth. Compare these bandwidth requirements with the typical 15 mbps bandwidth of a magnetic disk drive\(^2\), which is not expected to increase significantly in the near future [PGK88].

The solution proposed in this paper is to decluster the disk drives so that the aggregate bandwidth compensates the high bandwidth requirements of the multimedia devices. Theoretically, the clustering of \(d\) disk drives with a bandwidth requirement of \(B_{disk}\) each, could give a bandwidth of \(d B_{disk}\). For example, if we have a video following the NTSC standard (bandwidth requirement of 45 mbps) and the disk drives bandwidth is 15 mbps, we decluster the disk drives in groups of 3. In reality, we have to consider the activation, seek and latency time when computing the aggregate bandwidth. In fact, they increase as the size of the cluster increase (more devices need to be activated and more heads need to be repositioned).

Assuming that all the disk drives in the system are identical, and a database consisting of objects that belong to a single media type with bandwidth requirement \(B_{Display}\), we utilize the aggregate bandwidth of \(d\) disk drives to support a continuous display of an object. This is achieved as follows. First, the \(D\) disk drives in the system are partitioned into \(R\) disk clusters where \(R = \left\lceil \frac{D}{d} \right\rceil\). Next, each object in the database (say \(X\)) is stripped [SGM86] into \(n\) equi-sized subobjects\(^3\) \((X_1, X_2, ..., X_n)\). Each subobject \(X_i\) represents a contiguous portion of \(X\). When \(X\) is materialized from the tertiary storage device, its subobjects are assigned to the disk clusters in a round-robin manner, starting with an available cluster. In a cluster, a subobject is declustered [RE78, LKB87, GD90] into \(d\) pieces (termed fragments), with each fragment assigned to a different disk drive in the cluster.

Assuming that the bandwidth of each disk cluster is high enough that it can be multiplexed between \(U_{Cluster}\) requests, the available memory is partitioned into \(R \times (U_{Cluster} + 1)\) frames. To ensure a continuous display of an object, the system maintains a time cycle for each cluster. A time cycle consists of \(U_{Cluster}\) time intervals (also termed slots). A time interval is the time required for a cluster to reposition its disk heads and transfer a subobject into a memory frame. The duration of a time cycle corresponds to the display time of a subobject. Given a request for an object \(X\) that consists of \(n\) subobjects, the system reserves \(n\) time intervals on behalf of this request, one per time cycle of the system. Each reserved time slot occupies the same slot per time cycle. Relative in time, the distance between any two time slots reserved on behalf of a request is \(U_{Cluster} - 1\). The cluster employed in the first time cycle is the one containing \(X_1\) (say \(C_i\)). The display of \(X\) starts once \(X_1\) is staged in memory. In the second cycle, cluster \(C_{i+1 \mod R}\) is

---

\(^2\)The concepts described in this paper are applicable to other secondary storage devices.

\(^3\)\(X_n\) is an exception to this statement.
Figure 1: A schedule for servicing three requests

<table>
<thead>
<tr>
<th>Cluster 1</th>
<th>Cluster 2</th>
<th>Cluster 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>$X_2$</td>
<td>$X_3$</td>
</tr>
<tr>
<td>$X_4$</td>
<td>$X_5$</td>
<td></td>
</tr>
<tr>
<td>$Y_1$</td>
<td>$Y_4$</td>
<td>$Y_2$</td>
</tr>
<tr>
<td>$Y_3$</td>
<td>$Y_5$</td>
<td></td>
</tr>
<tr>
<td>$Z_2$</td>
<td>$Z_3$</td>
<td>$Z_4$</td>
</tr>
<tr>
<td>$Z_5$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: An assignment of subobjects to a three cluster system

employed to read $X_2$. $X_2$ becomes memory resident immediately before the display of $X_1$ completes because the duration of a cycle corresponds to the display time of a subobject ($X_1$ in this case). The system switches from the memory frame containing $X_1$ to $X_2$ to support a continuous display of $X$. The system iterates over the clusters and memory frames until $X$ is displayed in its entirety employing a single cluster in each time cycle.

To illustrate, assume a system that consists of 3 disk clusters. Moreover, assume that the bandwidth of each cluster is twice the bandwidth required to display an object ($U_{Cluster} = 2$). Let the following three objects reside on the disk clusters: $X$, $Y$, and $Z$. Each object is striped into 5 subobjects, and assigned to the clusters in a round-robin manner starting with a different cluster for each object (say cluster 1 for $X$, cluster 2 for $Y$, and cluster 3 for $Z$; the assignment of subobjects to the different clusters is shown in Table 1). Assume that three requests are issued, each referencing a different object with the following order of arrival: $X$, $Y$, followed by $Z$. Figure 1
demonstrates the scheduling of the time intervals to display the different subobjects as a function of time. This schedule is possible due to the assumed assignment, allowing the system to start the display $X$, $Y$ and $Z$ in the same time cycle. Note that $X_i$ employs the same time interval in each time cycle (interval 1). Moreover, relative in time, the slots reserved on behalf of a display (say $Y$) are $(U_{Cluster} - 1)$ slots apart. The duration of a time cycle corresponds to the display time of each subobject. Consequently, $X_2$ will be memory resident before the display of $X_1$ completes.

If the round-robin assignment of the objects started with the same cluster (say cluster 1) then $X_1, Y_1,$ and $Z_1$ would have been assigned to cluster 1. In this case, the display of $X$ and $Y$ would start in the first time cycle ($X_1$ occupying interval 1, $Y_1$ occupying interval 2 of cluster 1) while the display of $Z$ would employ a time interval in the second cycle of cluster 1. Thus, the duration of a time interval determines the wait time for the request referencing object $Y$, while the duration of a time cycle determines the wait time for the request referencing object $Z$. Note that the round-robin assignment of subobjects enables the system to support a maximum of six simultaneous displays4.

The concept of declustering a multimedia object across multiple disk drives in order to support its continuous display was originally described in [Ram92, GRAQ91, GR93]. These studies assumed a shared-nothing architecture [Sto86] as the hardware platform of a multimedia information systems. In [GS93], an extension with a tertiary storage device is assumed, in order to provide on-line access to vast amount of data. It introduced the concept of disk cluster and virtual data replication as a mechanism to support the display of objects to different users. But it does not strip the objects (i.e., the entire object resides in one disk cluster) and does not consider a mix of media types. In [BGMJ94], striping is used as an alternative to virtual data replication. With this technique, an object is striped into several subobjects, with each subobject assigned to a different disk cluster. It also assigns a degree of declustering to each bandwidth requirement of each media type based on the theoretical disk cluster bandwidth (i.e., ignoring the repositioning time of the heads). While these studies were a significant first step, they assumed a minimal amount of available memory and strived to establish a producer/consumer relationship between a disk cluster producing the data and a display stations consuming it (by displaying the data). This study extends the previous work by using memory as an intermediate staging area to display an object.

Since the size of multimedia objects is considerably large, compression is essential. But, when dealing with compressed objects the bandwith requirements of the displays become variable. Therefore, trying to synchronize the disk clusters to produce data at the same pace as the display stations consume it, could incur significant latency times. Because physical memory is not mechanical, staging data in memory before displaying it alleviates the problem.

4If all subobjects of $X$, $Y$ and $Z$ were assigned to cluster one (i.e., by ignoring the round-robin strategy), then cluster one could have displayed only two of the pending requests simultaneously while the third request would have had to wait (even though two clusters in the system remain idle waiting for work).
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>tfr</td>
<td>Transfer rate of a single disk drive</td>
</tr>
<tr>
<td>$T_{act}(d)$</td>
<td>Overhead of activating $d$ disk drives</td>
</tr>
<tr>
<td>$T_{seek}(d)$</td>
<td>Worst case seek time of $d$ disk drives</td>
</tr>
<tr>
<td>size(subobject)</td>
<td>Size of the unit of transfer between a disk cluster and the memory</td>
</tr>
<tr>
<td>size(mem)</td>
<td>Total size of memory</td>
</tr>
<tr>
<td>$B_{Display}$</td>
<td>Bandwidth required to display an object</td>
</tr>
<tr>
<td>$B_{Cluster}$</td>
<td>Bandwidth of a disk cluster</td>
</tr>
<tr>
<td>$B_{Tertiary}$</td>
<td>Bandwidth of the tertiary storage device</td>
</tr>
<tr>
<td>$D$</td>
<td>Number of disk drives in the system</td>
</tr>
<tr>
<td>$d$</td>
<td>Number of disk drives per disk cluster</td>
</tr>
<tr>
<td>$R$</td>
<td>Number of disk clusters in the system $R = \left\lfloor \frac{D}{d} \right\rfloor$</td>
</tr>
<tr>
<td>$U_{Cluster}$</td>
<td>Number of simultaneous displays supported by one disk cluster per time interval</td>
</tr>
<tr>
<td>$U_{I/O}$</td>
<td>Number of simultaneous displays supported by the I/O subsystem per time interval</td>
</tr>
</tbody>
</table>

Table 2: List of terms used repeatedly in this paper and their respective definitions

This paper address the following research topics:

- How many disk drives should be assigned to each cluster?
- What is the design of a cycle and its time interval, and how does it ensure a continuous display of a multimedia object?
- What is the size of a subobject?
- How does the system support a database that consists of a mix of multimedia objects, each with a different bandwidth requirement?
- How does the system support compression?

The rest of this paper is organized as follows. In Section 2, we describe the architecture assumed. Then we provide an answer to the above questions, starting with the simplest case and incrementally adding complexity. Section 3 focuses on a database that consists of non-compressed objects of a single media type and identifies the fundamental factors that impact a solution. Section 4 extends this work to a database that consists of a mix of media types. Section 5 extends this discussion to explain how compression can be implemented. Our conclusions and future research directions are contained in Section 6.
2 Architecture

This paper presents an approach to support the continuous display of objects based on a hierarchical storage architecture that consists of a tertiary storage device, a group of disk drives and memory.

The purpose of having a hierarchical architecture is to strike a compromise between the quality of the service and the cost of providing such a service. [MWS93] identifies the second factor as an important criteria for a multimedia information system. While memory is ideal for fast service, its cost and the massive information that a multimedia information system manages, makes secondary storage necessary. In addition, costwise tertiary storage is favorable over disk drives. But, the delays incurred because of the repositioning time of the read head of tertiary storage devices are higher than the ones incurred by the disk drives. Therefore, a hierarchical architecture enables the system to maximize the utilization of its resources in order to service many more requests simultaneously.

We assume that the entire database is stored in a tertiary storage device and the objects are materialized to the disk drives on demand. Then, the objects are staged into main memory to be later displayed. Buffering the data in memory could help to alleviate the differences of production (disk cluster bandwidth) and consumption (display bandwidth) rates without burning out disk drive bandwidth caused by repositioning the heads too often. The solution presented in this paper assumes that part of the memory is assigned in advance to serve as a buffer between the disk drives and display stations.

We consider two alternative organizations of the components in the hierarchical architecture: 1) memory serves as an intermediate staging area between the tertiary storage device, the disk drives and the display stations, and 2) the tertiary storage device is accessible only to the disk drives via a fixed size memory. With the first organization, the system may elect to display an object from the tertiary storage device by using memory as an intermediate staging area. With the second organization, the data must first be staged on the disk drives before it can be displayed. We capture these two organizations using three alternative paradigms for the flow of data among the different components:

- **Sequential Data Flow (SDF):** The data flows from tertiary to memory (STREAM 1 of Figure 2), from memory to the disk drives (STREAM 2), from disk drives back to memory (STREAM 3), and finally from memory to the display station referencing the object (STREAM 4).

- **Parallel Data Flow (PDF):** The data flows from tertiary to memory (STREAM 1), and from memory to both the disk drives and the display station in order to materialize (STREAM 2)
and display (STREAM 4) the object simultaneously. (PDF eliminates STREAM 3.)

- Incomplete Data Flow (IDF): The data flows from tertiary to memory (STREAM 1) and from memory to the display station (STREAM 4) to support a continuous retrieval of the referenced object. (IDF eliminates both STREAM 2 and 3.)

The tertiary storage device impacts: 1) Memory requirements only (ID), 2) Memory and disk bandwidth requirements (SDF, PDF). This paper focuses in the interaction between disk drives, memory and display stations. The first impact is studied in [GDS94]. The second impact can be viewed as an additional path (STREAMS 1 and 2). The difference with STREAMS 3 and 4 (the ones considered in this paper) is that the roles are changed, the disk drives become consumers instead of producers. But, the goal is to equate the producer and consumer rates therefore the change of roles is irrelevant for the configuration procedures. Hence, tertiary can be modeled as another media type.

### 3 Single Media Type

This section focuses on a database that consists of a single media type (i.e., the bandwidth requirement of each object, $B_{Display}$, is identical). Assuming that a system consists of $D$ disk drives and
a fixed amount of memory, we develop a technique that computes the configuration parameters of the system \( (d, \text{size}(\text{subobject})) \). The objective of this technique is to maximize the number of simultaneous displays. This enhances the overall performance of the system by improving the response time of the system and increasing its overall useful utilization.

We first discuss how to construct the disk clusters to support displays without hiccups under some constraints, and what the memory constraint is. Finally, we describe a configuration procedure and heuristic search to find \( d \) and \( \text{size}(\text{subobject}) \) that achieve the maximal \( U_{I/O} \).

### 3.1 Display Without Hiccups and Memory Constraint

The motivation for partitioning the \( D \) disk drives into \( R \) clusters is to increase the I/O bandwidth of the system: partitioning the disks increases the fraction of time the disk drives spend performing useful work (reading data) instead of wasteful work (either waiting to be activated or repositioning their heads). Consider the alternative forms of wasteful work in turn. First, the overhead of activating a disk cluster increases as a function of additional disk drives \( (d) \) that constitute a cluster, modeled as \( T_{\text{act}}(d) \). Second, the average seek time of a cluster increases as a function of \( d \) if the organization of data across the \( d \) disk drive is not identical. Once a request is activated on a cluster, its seek time is determined by the disk drive that has the longest seek time [LKB87, PGK88]. The expected seek time of a disk cluster that consists of \( d \) disk drives was derived in [BG88]. Assuming that \( T_{\text{act}}(d) \) is known, the bandwidth of a disk cluster as a function of \( d \) and the size of a subobject can be defined as:

\[
B_{\text{Cluster}} = \frac{\text{size}(\text{subobject})}{T_{\text{act}}(d) + T_{\text{seek}}(d) + \left( \frac{\text{size}(\text{subobject})}{d\text{fr}} \right)}
\]

where \( tfr \) is the transfer rate of a single disk drive. One constraint on the bandwidth of a cluster is that it should be greater than or equal to the bandwidth required to support a continuous display of an objects in the database:

\[
B_{\text{Cluster}} \geq B_{\text{Display}}
\]

Otherwise, the system would not be able to support a continuous display of an object.

If \( B_{\text{Cluster}} \) is significantly higher than \( B_{\text{Display}} \), then a disk cluster can be multiplexed among \( U_{\text{Cluster}} \) requests:

\[
U_{\text{Cluster}} = \frac{B_{\text{Cluster}}}{B_{\text{Display}}}
\]

In this case, the size of a subobject (say \( X_i \)) should be chosen such that its display time (i.e., \( \frac{\text{size}(\text{subobject})}{B_{\text{Display}}} \)) is greater than or equal to the sum of: 1) the amount of time a disk cluster is
Figure 3: A schedule for servicing three requests using a cluster.

multiplexed among \( U_{\text{Cluster}} - 1 \) other requests accessing subobjects\(^5\), and 2) the time required to read the next subobject \((X_{i+1})\). This constraint can be expressed as follows:

\[
\frac{\text{size}(\text{subobject})}{B_{\text{Display}}} \geq U_{\text{Cluster}} \times (T_{\text{act}}(d) + T_{\text{seek}}(d) + \frac{\text{size}(\text{subobject})}{d \times t_{fr}})
\] (4)

The reason using greater than or equal to is as follows. If the display time of a subobject is smaller than the right hand side of the inequality\(^4\), then the data will not be produced at the desired rate and will result in hiccups. When the display time of a subobject is greater, then the data will be produced faster than it can be consumed, hence we have to waste part of the disk bandwidth. Therefore we should aim for the equality.

For example, if all the subobjects of X, Y and Z are assigned to one cluster, \( U_{\text{Cluster}} \) is 3 and display time is 3.5 intervals (see Figure 3), then we will have to waste disk bandwidth. (i.e., disk cluster will be idle between the reading of Z1 and X2, Z2 and X3 ,and so forth)

Furthermore, since we want to maximize \( U_{\text{Cluster}} \) we use equality of \((4)\) to define \( U_{\text{Cluster}} \) as a function of \( d \) and size(subobject)\(^6\):

\[
U_{\text{Cluster}} = \frac{\text{size}(\text{subobject})}{B_{\text{Display}} \times (T_{\text{seek}}(d) + T_{\text{act}}(d) + \frac{\text{size}(\text{subobject})}{d \times t_{fr}})}
\] (5)

\(^1\)Note that the size of a subobject is fixed for all objects because this section assumes that all objects belong to a single media type.

\(^6\)\( U_{\text{Cluster}} \) is a real number, we will apply a floor function to make it to be an integer in our heuristic search.
or define \( \text{size}(\text{subobject}) \) as a function of \( d \) and \( U_{\text{Cluster}} \):

\[
\text{size}(\text{subobject}) = \frac{U_{\text{Cluster}} \times (T_{\text{seek}}(d) + T_{\text{act}}(d))}{B_{\text{Display}} - \frac{U_{\text{Cluster}}}{d_{\text{dfr}}}}
\]

These definitions are useful for explaining the search space of our optimization problem and the heuristic developed in this section.

The goal is to configure the system maximizing the total number of simultaneous displays, \( U_{\text{I/O}} \). We now define \( U_{\text{I/O}} \) as:

\[
U_{\text{I/O}} = R \times U_{\text{Cluster}}
\]

where \( R \) is the number of disk clusters in the system \( (R = \lfloor \frac{D}{U_{\text{Cluster}}} \rfloor) \). However, in order for the system to support \( U_{\text{I/O}} \) displays, the memory should consist of \( R \times (U_{\text{Cluster}} + 1) \) frames, where the size of a frame corresponds to the size of a subobject; hence, the following constraint:

\[
R \times (U_{\text{Cluster}} + 1) \times \text{size}(\text{subobject}) \leq \text{size}(\text{memory})
\]

### 3.2 Configuration

In this section, we describe a technique that strike a compromise between \( d \) and the \( \text{size}(\text{subobject}) \) to achieve maximal \( U_{\text{I/O}} \). Since we have two equations (4 and 8) and three variables \( (d, \text{size}(\text{subobject}) \) and \( U_{\text{Cluster}}) \), we approximate an optimal solution using a heuristic. We compute the maximal \( U_{\text{Cluster}} \) for each \( d \), then we can select the maximal \( U_{\text{I/O}} \) based on Equation 7.

The search space for this problem is shown from two different angles in Figure 4. This figure was generated assuming \( D=250, T_{\text{act}} = 2.5 \text{ microsec} \times d, B_{\text{Display}} = 45 \text{ mbps}, \text{tfr} = 28.8 \text{ mbps}, \) and \( \text{size}(\text{memory}) = 31.25 \text{ Gigabytes} \)

From Equation 5, it is easily to see that for given a fixed value of \( d \), \( U_{\text{Cluster}} \) increases as \( \text{size}(\text{subobject}) \) increases. As you can see in Figure 5.a, for given a fixed value of \( d \), the number of simultaneous displays \( (U_{\text{I/O}}) \) as a function of the \( \text{size}(\text{subobject}) \) increases in a step-wise manner. The height of each step is \( R \). The explanation of this is as follows. \( U_{\text{Cluster}} \) is a function of the size of the

\footnote{\text{size}(\text{subobject}) \text{ is a real number, therefor we have to align \text{size}(\text{subobject}) to the byte, we can do this alignment by interleaving way, for example, if \text{size}(\text{subobject}) is 127.3 bytes the we read 128 bytes in the first cycle, 127 bytes in the second and third, 128 in the fourth and so forth.}}

\footnote{Plus one per cluster because an additional frame is needed to momentarily stage both \( X_i \) and \( X_{i+1} \) in memory when the system switches between two subobjects.}

\footnote{In the current workstation technology, the ratio of memory size to disk capacity is \( \frac{1}{32} \), so for the given disk capacity, 250 * 4Gigabytes, we assume the memory size is 31.25 Gigabytes.}
Figure 5: Two dimensional view of the search space.
Figure 6: Eliminate search space using the memory constraint.

size(subobject) and should be an integer (so we apply a floor function on Equation 5 here). As one increases the size of a subobject, the value of $U_{Cluster}$ increases by one in regular intervals. Each time this happens, $U_{I/O}$ increases by $R$ (see Equation 7). Therefore, for each $d$, it suffices to compute the maximal size(subobject) that satisfies the constraints to obtain the maximal $U_{Cluster}$.

Unlike the case of a fixed $d$, if we fix the size(subobject), $U_{I/O}$ is not necessary a monotonic function with respect $d$. As it could be seen in Figure 5.b.

When memory constraint is violated due to limited amount of memory, one can satisfy it by striking a compromise between $d$ and the size(subobject) (these two parameters determine $U_{Cluster}$, and $U_{I/O}$ in turn, see Equation 5).

We change the inequality 8 into equality in order to compute the upperbound for $U_{Cluster}$. By combining Equation 4 and 8, we get $U_{Cluster}$ as a function of size(memory) and $d$:

$$U_{Cluster} = -b + \sqrt{b^2 + 4 \times ((T_{act}(d) + T_{seek}(d)) \times \frac{D}{4} \times tfr \times d) \times (\frac{size(mem)}{R_{display}} \times d \times tfr)}$$

(9)

where $b = (T_{act}(d) + T_{seek}(d)) \times \frac{D}{4} \times tfr \times d + size(mem)$.

The constraint posited in Equation 8 eliminates a portion of the search space (see Figure 6). This search space can be restricted further using the hardware characteristics of the magnetic disk drives: A magnetic disk drive is almost always required to reposition its head if the unit of transfer is larger than the size of a cylinder. Consequently, there are marginal advantages to choosing a size(subobject) that renders the unit of transfer from each disk drive to be larger than
Input: \( B_{Display}, D, \text{size(cylinder)}, tfr, T_{seek}(d), T_{act}(d) \) and size(memory)

Output: \( d, \text{size(subobject)}, \) and \( U_{I/O} \)

Begin
\[
d = \left\lceil \frac{B_{Display}}{tfr} \right\rceil \quad \text{/* \( d \) cannot be smaller than this value */}
\]
\( ProspectiveSet = \text{NULL} \)

For all \( (d \in \text{Integer}) \) and \( (\left\lceil \frac{B_{Display}}{tfr} \right\rceil \leq d \leq D) \) Do

\[
\text{size(subobject)} = d \times \text{size(cylinder)}
\]

Compute \( U_{Cluster} \) using Equation 5 and take the floor of its value

Compute \( \text{size(subobject)} \) using Equation 6

/* Check the memory constraint (Equation 8) */

If \( ((U_{Cluster} + 1) \times \left\lceil \frac{D}{d} \right\rceil \times \text{size(subobject)} > \text{Size(memory)}) \) then

Compute \( U_{Cluster} \) using Equation 9 and take the floor of its value

Compute \( \text{size(subobject)} \) using Equation 6

EndIf

Compute \( B_{Cluster} \) using Equation 1

If \( B_{Cluster} \geq B_{Display} \) then /* check the bandwidth constraint (Equation 2) */

Compute \( U_{I/O} \) using Equation 7

Insert \( (d, \text{size(subobject)}, U_{I/O}) \) into \( ProspectiveSet \)

EndIf

Increment \( d \) by one

End For

Choose the element of \( ProspectiveSet \) with the maximum \( U_{I/O} \) value and return its \( d, \text{size(subobject)} \) and \( U_{I/O} \).

End

Figure 7: A heuristic to compute \( d, \text{size(subobject)}, \) and \( U_{I/O} \)
Thus, we can limit the search by assuming that the size of a subobject is smaller than $d \times \text{size}(\text{cylinder})$. Figure 7 outlines a heuristic that employs this rule of thumb and performs an exhaustive search of the remaining space. It visits the different states by analyzing all possible values of $d$ that range from $\left\lceil \frac{B_{\text{display}}}{t_{\text{fr}}} \right\rceil$ (its lower bound) to $D$. For each value, it sets the size of a subobject as a function of $d$ and the cylinder size. Next, it uses Equation 5 to compute the number of users supported by a cluster ($U_{\text{cluster}}$) and convert this value to an integer. It uses $d$ and the obtained $U_{\text{cluster}}$ to recompute the size(subobject). This recomputation is necessary because the equality of 4 is no longer true for the new value of $U_{\text{cluster}}$, and it may reduce size(subobject) significantly. Using the obtained $d$, size(subobject), and $U_{\text{cluster}}$, the heuristic checks the memory constraint. If this constraint is violated, it recomputes $U_{\text{cluster}}$ as a function of $d$ and the available memory in order to compute a new size(subobject). It employs the obtained values to check the constraint posited in Equation 2. If they violate this constraint, then the obtained values are ignored from further consideration. Otherwise, they are treated as a solution and maintained in a set. This procedure is repeated for all values of $d$ upto $D$. From the obtained set of solutions, we choose the solution that maximizes $U_{1/0}$. The order of this heuristic is $\left(D - \left\lceil \frac{B_{\text{display}}}{t_{\text{fr}}} \right\rceil \right)$.

4 Mix of Media Types

We present a new approach to configure a system with a mix of media types. As compared to a database that consists of a single media type, the following additional factors impact the performance of the system: 1) the mix of media, 2) the scheduling policy and 3) the duration of each display. The approach presented, considers a representative queue of requests and a scheduling policy to compute the size of a subobject and the number of disk drives per cluster in order to maximize the performance of the system.

The model used to display the objects is similar to that of a database consisting of a single media type. The basic difference is that instead of reading subobjects, the system reads blocks. The size of a block depends on the bandwidth requirement of its object (i.e., the media type). The higher the bandwidth the larger the block size.

An immediate question is how to determine the size of a block. A naive approach is to configure the block size based on one media type (say A) and define the block size for the other media types (say B) as a function of this size (by multiplying the block size with $\frac{B_{\text{display}}(B)}{B_{\text{display}}(A)}$). We start with an example that demonstrates the configuration parameters produced by considering a single media system is not optimal from a global perspective. Subsequently, we describe the model and the new factors to be considered. Finally we present the algorithm to compute the optimal configuration parameters and present a case study where a heuristic can be applied to compute the parameters.
faster.

Example 4.1: Consider a system with 250 disks of 4 GBytes each and 31.25 GBytes of memory. Each disk has 3055 cylinders of approximately 1.3 MBytes each, and their activation time is 0.0000025 seconds. Also, the disk transfer rate is 28.8 megabits per second.

Assume a database consisting of five media types: CD audio, video using CIF, NTSC, CCIR Recommendation 601 and HDTV representations. Suppose that every display has a video and an audio component, therefore CD objects occur half of the time. Furthermore, the bandwidth required to display the objects of each media type, the heat\(^{10}\) of each media type and the average display time of the objects of each media type are as follows:

<table>
<thead>
<tr>
<th>Media Type</th>
<th>Bandwidth (mbps)</th>
<th>Heat (percentage)</th>
<th>Average Display Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>1.5</td>
<td>50</td>
<td>62</td>
</tr>
<tr>
<td>CIF</td>
<td>36</td>
<td>20</td>
<td>55</td>
</tr>
<tr>
<td>NTSC</td>
<td>45</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>CCIR</td>
<td>216</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>HDTV</td>
<td>836</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

Assume that the objects are referenced randomly. If we build a representative queue of 800 requests\(^{11}\) that satisfy the previous assumptions, we can compare the service time of the queue when the system is configured based on a single media type with the service time when configured based on the representative queue of requests.

The following table shows the configuration parameters for each media type and the time required to service the representative queue:

<table>
<thead>
<tr>
<th>Bandwidth (mbps)</th>
<th>d disks</th>
<th>R clusters</th>
<th>size (subobject) (MB)</th>
<th>B(_{\text{Cluster}}) (mbps)</th>
<th>Service Time (hours : minutes : seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>1</td>
<td>250</td>
<td>0.875</td>
<td>25.85</td>
<td>Cannot support all the media types</td>
</tr>
<tr>
<td>CIF</td>
<td>13</td>
<td>19</td>
<td>16.25</td>
<td>324.08</td>
<td>Cannot support all the media types</td>
</tr>
<tr>
<td>NTSC</td>
<td>31</td>
<td>8</td>
<td>39.5</td>
<td>765.34</td>
<td>Cannot support all the media types</td>
</tr>
<tr>
<td>CCIR</td>
<td>35</td>
<td>7</td>
<td>45</td>
<td>864.03</td>
<td>6:43:02</td>
</tr>
<tr>
<td>HDTV</td>
<td>35</td>
<td>7</td>
<td>36.5</td>
<td>836.21</td>
<td>7:10:58</td>
</tr>
</tbody>
</table>

However, if the system is configured based on a representative queue of requests, the average service time is 5:30:41. Therefore, the service time when configuring based on a single media type could be 21.88\% or 30.33\% higher than when based on the representative queue. □

\(^{10}\)the probability that a request is of a given type

\(^{11}\)800 requests is large enough to make the requests compete for the system resources.
4.1 Model

This section presents a model to support simultaneous display of a mix of media types in a clustered multidisk system. We start by considering the new factors that impact the performance of the system. Subsequently, we describe how the objects can be displayed without hiccups and establish the memory requirements of the system. And finally, we list the parameters needed to support a mix of media types and their constraints.

4.1.1 New Factors

The factors that impact the configuration parameters of the system include: The mix of media, the scheduling policy, and the display time of the requests. Consider each in turn.

The first and obvious difference is that each media type has its own bandwidth requirement. This leads us to read the objects in blocks instead of subobjects. The size of the block is fixed for objects of one media type and is proportional to the bandwidth required by that media type. The size of a block is different for alternative media types because the consumption rates for the display of different media types is variant. Therefore, the amount of memory allocated for the display of each media type should be proportional to its consumption rate. If we allocate the same amount of memory for each media type, the display of an object with the lowest consumption rate would reside in memory longer than required, preventing the display of other objects. Therefore, for each bandwidth we use a different block size.

For the same reasons that we read only one subobject of an object per cycle, we read only one block of an object per cycle. And as before, the blocks are assigned to the clusters in a round robin manner. Also, we allow to read the blocks in pieces spread within a cycle. Otherwise, if the blocks must be read contiguously, it will limit the possible requests to fit in a cycle (we may have two or more non-contiguous time intervals that could be used to read a block). But it is important that the blocks are read in the same manner in all cycles\(^\text{12}\). That is, if an object \(A\) is split into blocks \(A_1 \ldots A_m\) and block \(A_1\) is split into pieces \(A_{11} \ldots A_{1p}\), then if the piece \(A_{1j}\) is read from \(t_{j}^{s}\) to \(t_{j}^{e}\) in a cycle, then for each \(i\), \(A_{ij}\) is read from \(t_{j}^{s}\) to \(t_{j}^{e}\) in the cycle where \(A_i\) is read. The following example illustrates a possible scheduling of the display of an object.

Example 4.2: Assume that an object \(A\) consists of 3 blocks and given the current status of the system, the system is forced to split each block into 3 pieces. Suppose that we configure the system with 5 clusters. A possible schedule to the object is:

\(^{12}\)Otherwise hiccups could be caused or pieces of the object would reside in memory longer than required
At cycle $i$, cluster $j$ is scheduled as:

```
<table>
<thead>
<tr>
<th></th>
<th>A11</th>
<th></th>
<th>A12</th>
<th></th>
<th>A13</th>
</tr>
</thead>
</table>
0  | $t_1^i$ |   | $t_2^i$ |   | $t_3^i$ |
```

At cycle $i+1$, cluster $(j+1) \mod 5$ is scheduled as:

```
<table>
<thead>
<tr>
<th></th>
<th>A21</th>
<th></th>
<th>A22</th>
<th></th>
<th>A23</th>
</tr>
</thead>
</table>
0  | $t_1^i$ |   | $t_2^i$ |   | $t_3^i$ |
```

At cycle $i+2$, cluster $(j+2) \mod 5$ is scheduled as:

```
<table>
<thead>
<tr>
<th></th>
<th>A31</th>
<th></th>
<th>A32</th>
<th></th>
<th>A33</th>
</tr>
</thead>
</table>
0  | $t_1^i$ |   | $t_2^i$ |   | $t_3^i$ |
```

The mix of media impacts the performance of the system. The order in which different media types are queued in a system that serves on a first come first served basis, could affect the performance. The following example illustrates this effect.

**Example 4.3:** Suppose that the database consists of objects that belong to two different media types, A and B. $B_{display}(A) = 10$ mbps while $B_{display}(B) = 20$ mbps. Suppose that a cycle consists of 5 time slots$^{13}$, the block sizes of A and B are 1 and 2 subobjects respectively, and there is only one disk cluster. Suppose that the scheduler serves the requests in a FCFS manner and a queue of 20 requests is waiting to be served. Assume that the size of each referenced object is 1 block. Moreover, suppose that the queue starts with a request of A followed by a request of B and every other request is of the same type. Therefore, the scheduler will read three requests during each cycle for the first six cycles and then the last two requests in the seventh cycle.

If instead, the queue consists of a request of A followed by two requests of B, with the same pattern repeated 4 times, followed by 5 requests of A. Then, the only difference with the previous

$^{13}$ Even though, we are reading blocks of objects instead of subobjects, we maintain the concept of subobject as a unit of reading to consider the repositioning time.
case is the order of the requests. However, the scheduler requires only 6 cycles to service the queue. □

As shown in the following example, another factor that impacts the performance of the system is the scheduling policy.

Example 4.4: Suppose that we have a situation similar to the first case of Example 4.3. But, the scheduler selects the next request in the queue that fits in what is left in the cycle. Then, we can schedule to service the requests 1, 2, 3 and 5 in the first cycle, 4, 6, and 7 in the second cycle and so forth. Therefore, only 6 cycles are required to service the queue. Which is shorter than the 7 cycles required with the first-come-first-served policy of Example 4.3. □

The other factor to consider is the duration of each display. As demonstrated in Example 4.3, the way the media types are mixed affect the performance of the system. Increasing the length of a request can be represented by adding an extra request, of length equal to the increase in size, after the original request. For example: If the original queue consists of two requests of different media types, and the first request size is duplicated. Then, we can represent the new queue as consisting of three requests, with the second request as a request identical to the first request (the first and the last request is the same as the original queue). Since the mix of media is different in the second case, its performance could be different.

We thus configure the system based on a representative queue of requests, which captures the factors described above.

4.1.2 Displaying without hiccups

We now describe how to display objects without hiccups. Since the display of a block takes a cycle and all the blocks are read in the same way in all the cycles, it suffices to consider how to display one block without hiccups.

Assume that we split the cycle in *equi-intervals*, time intervals of equal duration $t$ such that

$$t \leq \frac{\text{size(subobject)}}{B_{\text{cluster}}}.$$  \(^{14}\)

Also, assume that the block $b$ is read in a cycle as follows:

```
0 \hspace{0.5cm} t_1^b \hspace{0.5cm} t_1^e \hspace{0.5cm} t_2^b \hspace{0.5cm} t_2^e \hspace{0.5cm} \ldots \hspace{0.5cm} t_j^b \hspace{0.5cm} t_j^e \hspace{0.5cm} \ldots \hspace{0.5cm} t_m^b \hspace{0.5cm} t_m^e
```

Where $b$ is the concatenation of the pieces $b_1, \ldots, b_m$. And, each piece lies within one equi-interval.

\(^{14}\)Time to read a subobject.
Intuitively, the algorithm to schedule the display of the block $b$ tries to start the display at $t_i^*$, if there is a hiccup, it tries to fix it by starting later on, so that the hiccup produced before will not happen. And continues until it finds a beginning time when there will not be hiccups. It considers only the starting times such that $b_j$ is displayed starting at $t_j^*$, for some $j$. More precisely:

**Input:** $|b_1|, \ldots, |b_m|, t_1^*, \ldots, t_m^*$

**Output:** Time $t$ at which we can start displaying $b$

**Begin**

$\text{found} := \text{false}$

$i = 1$

While $\neg \text{found}$

Let $t$ be the time to start displaying $b$ such that the display of $b_i$ starts at $t_i^*$

If There is no hiccup starting the display of $b$ at $t$

Then $\text{found} = \text{true}$

Return $t$

Else Let $i$ be the smallest $j$ such that if the display of $b$ starts at $t$

then there is a hiccup before the display of $b_j$ (i.e. $b_j$ is not ready in memory when we finish displaying $b_{j-1}$)

End If

End While

End

We can assure that the previous algorithm will always terminate. First, notice that after setting $t$ in the loop, the pieces $b_1, \ldots, b_i$ can be displayed without hiccups. Therefore, the values of $i$ increases as the number of iterations of the loop. But the maximum value of $i$ is $m$, therefore the number of iterations is bounded.

Since the display time of a block requires a cycle, we start displaying the following block at $t$ in the next cycle. Furthermore, we will not have hiccups because we were able to display $b$ without hiccups and the distribution of the pieces in a cycle is exactly the same as $b_1, \ldots, b_m$ in the current cycle.

In conclusion, the previous algorithm will always give us a way of scheduling the display of an object without hiccups.
4.1.3 Memory Requirement

The use of memory as an intermediate stage gives us flexibility to handle different consumption rates. Since the number of disk drives per cluster is fixed, the production rate is fixed. While the consumption rate varies with the media type of the object. Another advantage of having memory is to be able to read a block anywhere within the cycle.

We now establish the memory requirement for a given configuration. We start by describing how memory is managed and then we prove that we require at most \( R \cdot (n + 1) \cdot \text{size}(\text{subobject}) \) as the size of the memory, where \( n \) is the length of the cycle in time slots.

If \( n \) is an integer, we can split the memory in pages of size equal to \( \text{size}(\text{subobject}) \). The pages are allocated to disk clusters and time intervals as follows: For each interval \( i \) of the cluster \( j \) in the cycle \( k \), allocate page number \((k \cdot n + i) \mod (n + 1) + [(j - (k \mod R)) \mod R] \cdot (n + 1)\). For example, assume that \( n = 3 \) and \( R = 2 \). Then the memory assignment is:

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 7</th>
<th>Page 4</th>
<th>Page 5</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 1</td>
<td>Page 4</td>
<td>Page 5</td>
<td>Page 6</td>
<td>Page 3</td>
<td>Page 0</td>
<td>Page 1</td>
<td>Page 6</td>
<td>Page 7</td>
<td>Page 4</td>
</tr>
</tbody>
</table>

If \( n \) is not integer, we can split the cycle in equi-intervals instead of time intervals and the page size to be \( B_{\text{Cluster}} \cdot \text{equi-interval} \) instead of \( \text{size}(\text{subobject}) \). Then the memory requirement is 
\( R \cdot n \cdot \text{size}(\text{subobject}) + R \cdot B_{\text{Cluster}} \cdot \text{equi-interval} \). Since an equi-interval is shorter than a time interval, the memory requirement is at most 
\( R \cdot (n + 1) \cdot \text{size}(\text{subobject}) \).

Finally, we prove that we require at most 
\( R \cdot (n + 1) \cdot \text{size}(\text{subobject}) \) of memory to schedule the display of objects without hiccups. By proving that the memory management just described can be used by the algorithm in Section 4.1.2.

**Lemma 4.5:** If the length of the cycle is \( n \) then we need at most \( R \cdot (n + 1) \cdot \text{size}(\text{subobject}) \) of memory to be able to display an object without hiccups.

**Proof:** Assume that we schedule the display following the algorithm in Section 4.1.2. Then we prove that the memory management described above is appropriate for the scheduling algorithm.

It suffices to show that for each \( i \), the memory allocated to read \( b_i \) in cycle \( p \) is released by \( t_i \) in the cycle \( p + 1 \).

Let \( j \) be the smallest index such that the display of \( b_j \) starts at \( t_j \). It is easy to see that such \( j \) exists. We consider two cases:
$i < j$: The memory used is released when the reading of $b_j$ finishes. Therefore, it is available for the next cycle.

$i \geq j$: Since the block is supposed to be displayed in a cycle, the memory used for $b_j, \ldots, b_m$ is released by $t$ in the next cycle. Since $t \leq t_j^*$, the memory used for $b_j, \ldots, b_m$ is available by $t_j^*$.

\[\square\]

### 4.1.4 Parameters and Constraints

We now list the parameters of the model and their constraints. Given $k$ types of media, each with a bandwidth requirement of $B_1, \ldots, B_k$, the following parameters determine the performance of the system.

(i) $n$: The length of a cycle\(^{15}\).

(ii) $\text{size}(\text{subobject})$: The size of a subobject.

(iii) $d$: The number of disk drives per cluster ($R = \lceil \frac{D_j}{d} \rceil$).

(iv) $n_1, \ldots, n_k$: The size of a block for each media type.

To implement the model described above, the parameters must satisfy the following constraints:

(i) A block is read within a cycle. For each $i$, $n_i \leq n$.

(ii) The display time of a block is a cycle. For each $i$:

\[
\frac{n_i \times \text{size}(\text{subobject})}{B_i} = \frac{n \times \text{size}(\text{subobject})}{B_{\text{Cluster}}} \tag{10}
\]

(iii) The memory requirement can be satisfied,

\[R \times \text{size}(\text{subobject}) \times (n + 1) \leq \text{size}(\text{mem}) \tag{11}\]

(iv) On the average, the size of each fragment is less than one cylinder,

\[\text{size}(\text{subobject}) \leq Q \times d \tag{12}\]

\(^{15}\)In time slots units.
The disk cluster bandwidth can support the display of a subobject without hiccups,

\[ B_{\text{Cluster}} \geq \max\{B_1, \ldots, B_k\} \]  

(13)

Constraint posited in Equation (10) can be expressed as:

\[ \frac{n_1}{B_1} = \frac{n_2}{B_2} = \cdots = \frac{n_k}{B_k} = \frac{n}{B_{\text{Cluster}}} \]

Notice that \( n, n_1, \ldots, n_k \) are real numbers, therefore we have to align the block sizes to the byte. The alignment can be done by interleaving alignment to the higher byte and to the lower byte. For example, if the block size is 127.3 then we read 128 bytes in the first cycle, 127 bytes in the second and third, 128 in the fourth and so forth.

We assumed that the heads are repositioned once for each time slot. That assumption can be changed easily by modifying the seek time in the computation of \( B_{\text{Cluster}} \). For example, if on average the heads are repositioned twice in an interval, we can multiply the seek time in \( B_{\text{Cluster}} \) by a factor of 2.

4.2 Configuration

In this section we present an algorithm to compute the parameters of the model in order to minimize the service time. This algorithm is based on a representative queue of requests.

We start by reducing the parameters to consider only \( \text{size}(\text{subobject}) \) and \( d \). Subsequently, we present the algorithm to compute the optimal parameters. Next, we consider a case study and describe a heuristic to make the computation faster.

The previous section demonstrated that the configuration parameters are: \( \text{size}(\text{subobject}) \), \( d \), \( n, n_1, \ldots, n_k \). We now show how the number of parameters can be reduced to two, namely, \( \text{size}(\text{subobject}) \) and \( d \). Given a size of a subobject and the number of disk drives per cluster, we can compute \( B_{\text{Cluster}} \). For a given \( B_{\text{Cluster}} \), Constraint (10) limits the number of choices for \( n, n_1, \ldots, n_k \). In particular, if \( n^0, n_1^0, \ldots, n_k^0 \) satisfies (10) then the \((k+1)\)-tuples that satisfy the constraint can be expressed as \( c * (n^0, n_1^0, \ldots, n_k^0) \), where \( c \) is a scalar. Therefore, the different values for the \((k+1)\)-tuple does not affect the service time. For example: if \( c = 2 \) then with \( c * (n^0, n_1^0, \ldots, n_k^0) \) instead of \( (n^0, n_1^0, \ldots, n_k^0) \) we would have cycles twice as long, however, we also read blocks that are twice in size. Thus, the net effect is the same. Hence, it suffices to consider one possible value for \( n_1, \ldots, n_k, n \), that satisfies constraint (10) for a given value of \( B_{\text{Cluster}} \).

The selection of the \((k+1)\)-tuple is made to minimize the fragmentation of blocks in a cycle. When a request finishes, it leaves a fragment of time that can be used for the next request. But
if the next request does not fit exactly in the fragment, we end up leaving smaller fragments to be used later on. This could lead to fragmentation of the blocks within a cycle, in pieces smaller than a subobject. Therefore, we decided to select the \((k+1)\)-tuple that gives us the largest block sizes. Given a value for \(B_{\text{Cluster}}\), we select the value for \(n_1, \ldots, n_k, n\), considering the largest \(n\) that satisfies constraint (11).

The following algorithm computes the configuration parameters. We assume that there is a function \(\text{ServTime}(\cdot)\), which gives the service time of a queue of requests for a given configuration. \(\text{ServTime}(\cdot)\) computes the service time based on the scheduling policy.

**Input:** \(B_1, \ldots, B_k, D, Q, \text{size(mem)}, \text{Queue}, \text{ServTime}(), \text{mins}, \text{maxs}, \text{deltas}, \text{mind}, \text{maxd}, \text{deltad} \)

**Output:** \(\text{size(subobject)}, d\)

**Begin**

\(\text{minST} = \text{Time to service Q u e e in a sequential manner (one at a time)}\)

\(\text{bests} = \text{mins}\)

\(\text{bestd} = \text{mind}\)

**For** \(s = \text{mins}, \text{maxs}, \text{deltas}\)

**For** \(d = \text{mind}, \text{maxd}, \text{deltad}\)

\(\text{/* Compute the length of the cycle */}\)

\(n = \frac{\text{size(mem)}}{\left\lfloor \frac{s}{\text{size(subobject)}} \right\rfloor} - 1\)

\(\text{/* Check constraints */}\)

**If** \(n > 0\) **and** constraint (12) is not violated **and** constraint (13) is not violated

**Then** \(\text{/* Compute the size of the blocks for each media type */}\)

**For** \(i = 1, k\) **Do** \(n_i = \frac{n}{B_{\text{Cluster}}[\text{size(subobject)}, d]} + B_i\)

\(\text{CurrentST} = \text{ServTime(Queue, n, n_1, \ldots, n_k, s, d)}\)

**If** \(\text{CurrentST} < \text{minST}\)

**Then** \(\text{minST} = \text{CurrentST}\)

\(\text{bests} = s\)

\(\text{bestd} = d\)

**End If**

**End If**

**Next** \(d\)

**Next** \(s\)

**End**

How expensive the computation is, depends on the search space. Which is the values of \(\text{size(subobject)}\) and \(d\) for which \(\text{ServTime}(\cdot)\) is invoked. It can potentially be invoked, \(D *\)
size(mem) times. But constraint (12) reduces it to $\frac{Qd^2}{2}$ times. Which is reduced further by constraint (13).

We conclude this section by presenting a case study where a heuristic can reduce the computation time, namely, a system that serves the requests on the first come first served basis. We first demonstrate that, if we disregard the constraints, then $\text{ServTime}$ for a given $d$ decreases as size(subobject) increases. We thus need to consider only the largest size(subobject) that satisfies constraints (11), (12) and (13), for each $d$. Which reduces the potential search space to $1, D$.

**Lemma 4.6:** Assume that the system serves the requests on a first come first served basis. Then for a given $d$, if $s_1 < s_2$ and $(s_1, d)$ and $(s_2, d)$ satisfy constraints$^{16}$ (11), (12) and (13) then $\text{ServTime}(\text{Queue}, n, n_1, \ldots, n_k, s_1, d) \geq \text{ServTime}(\text{Queue}, n, n_1, \ldots, n_k, s_2, d)$

**Proof:** As the selection of the duration of a cycle potentially does not affect the service time, we fix the length of a cycle to $\tau$ for both cases: $s_1$ and $s_2$. Where $\tau$ satisfies constraints$^{17}$ (11), (12) and (13) for both $(s_1, d)$ and $(s_2, d)$. Then we compare the service time when the cycle length is $\tau$ for both $(s_1, d)$ and $(s_2, d)$.

We first select $\tau$ to be the duration of the cycle computed by the algorithm for $(s_2, d)$.

$$\tau = \left( \frac{\text{size(mem)}}{\left\lfloor \frac{D}{d} \right\rfloor * s_2} - 1 \right) * (T_{act} * d + T_{seek} + \frac{s_2}{lfr * d}).$$

Since,

$$\left( \frac{\text{size(mem)}}{\left\lfloor \frac{D}{d} \right\rfloor * s_1} - 1 \right) * (T_{act} * d + T_{seek} + \frac{s_1}{lfr * d}) > \tau$$

The length of the cycle computed by the algorithm for $(s_1, d)$ is greater than $\tau$.

Therefore, when considering $(s_1, d)$, the number of intervals in each cycle $\tau$, is smaller that the number computed by the algorithm:

$$n = \frac{\tau}{B_{\text{Cluster}}(s_1, d)} < \left( \frac{\text{size(mem)}}{\left\lfloor \frac{D}{d} \right\rfloor * s_1} - 1 \right)$$

Which implies that $n$ satisfies constraint (11) for $(s_1, d)$. Therefore, we can consider $\tau$ as the length of the cycle to compute the service time for both, $(s_1, d)$ and $(s_2, d)$.

For a given media type $i$, the size of the block (in bytes) read for $(s_1, d)$ and $(s_2, d)$ is $\tau * B_i$. Therefore, the number of cycles taken to display an object is the same for both cases, $(s_1, d)$ and $(s_2, d)$.

---

$^{16}$A pair $(s, d)$ satisfies constraint (11) if there exists a positive number $n$, such that the constraint is satisfied for $(s, d, n)$.

$^{17}$Notice that the values $n, n_1, \ldots, n_k$ can be uniquely determined by $\text{size(subobject)}$, $d$ and the length of the cycle.
Since $B_{\text{cluster}}(s_1, d) < B_{\text{cluster}}(s_2, d)$ and $\lceil \frac{D}{T} \rceil$ is the same for both cases. We could be able to schedule the display of an object for $(s_2, d)$, earlier than for $(s_1, d)$. And furthermore, there is no object that is scheduled for $(s_1, d)$ earlier than for $(s_2, d)$. Therefore, the service time for $(s_2, d)$ is smaller or equal to the service time for $(s_1, d)$. \(\square\)

### 5 Compression

Compression could make the system use its resources more efficiently (i.e., memory and disk bandwidth). We study how to incorporate compression to display objects under the architecture described in Section 2. The only assumption about the compression technique is that it allows the display of data in an incremental manner (i.e., we can start the display of an object with the data that is currently in memory. We do not have to wait for additional data to start the display). Therefore, our approach is general enough to cover a considerable number of compression techniques currently in use.

The order in which compression is done with respect to the stripping determines the behavior of the system. We present one model for each of the following orderings: 1) Compress the object before splitting it into blocks, and 2) Split the object into blocks before compressing. And describe how to configure the system for each case. In fact, the configuration procedures for each model can be done using the configuration for non-compressed objects with the appropriate parameters.

#### 5.1 Compress then Split

The average bandwidth requirement for a compressed image is potentially smaller than for a non-compressed image. If we consider the model presented in Section 4.1, the display time of a block is one cycle. Therefore, the display time of a compressed block could be potentially longer than a cycle. If we compress the objects before splitting them into blocks, we may have some extra disk bandwidth available for other applications without any lost in display throughput. For example, if the average ratio for image compression is 10 to 1, then the average display time of a block for a compressed image is ten cycles while the display time of a non-compressed block is one cycle. Therefore, the disk bandwidth assigned to an object can be used for other applications during 9 cycles (in average).

On the other hand, if the system is used exclusively for the display of objects, we may want

---

\(^{18}\) We are taking a cycle as the unit of time. We are not considering the specific time within the cycle that the display starts.
to use the extra disk bandwidth to service other objects. But, since the duration of the display of a block is variable the concept of cycle not longer applies. We still have to fragment the time in intervals for each disk cluster. Only one request can be read during one time interval. The duration of those time intervals could be either fixed or variable. Another factor to consider is the the size of a block, it could be fixed or variable.

We present two models to schedule requests when the objects are compressed and then split into blocks. One approach considers time intervals of variable duration and block sizes proportional to the average bandwidth requirement of its media type. The other considers time intervals of fixed duration and blocks of fixed size. Then we present a method to configure the system for either approach.

5.1.1 Variable Time Intervals and Variable Block Sizes

The advantage of making the block size proportional to the bandwidth requirement of its media type is to use the memory as fair as possible. For example when the objects are not compressed, in average, each unit of memory will hold a fraction of an object for the same amount of time (one cycle). With that goal in mind, considering block sizes proportional to the average bandwidth requirement of its media type is a good approach. We start by giving a brief description of the model, then the scheduling algorithm. Next, we revise the constraints posited in Section 4.1.4.

The idea is to read block by block of a request from the proper disk cluster without violating memory constraints and making sure that there is a way of scheduling the display without hiccups. The only difference with the model presented in Section 4.1 is that we do not have to read a block within a cycle. Also, the way blocks are read does not have to follow any pattern timewise. Thus, the scheduler serves a request based on the availability of resources and real time constraints (no hiccups), by assigning time intervals to the request.

The scheduler needs to keep track of the disk bandwidth available, by marking the time intervals already assigned to a request for each disk cluster. It will assign non-contiguous time intervals to a block, if necessary. To assure that memory is sufficient to service a request, the scheduler must keep track of the memory available. To trace memory utilization, it splits the memory into frames and start assigning frames to requests as they are scheduled. To release a frame, it computes the time when the data in the frame has been completely displayed. Such computation requires the following information: Time when data in the frame starts being displayed, and average\textsuperscript{19} compression factor. When more than one object is using the same frame, it considers the last

\textsuperscript{19}The compression factor can vary within a frame. Without loss of generality, we can assume the average rate to compute the duration of the display.
object that finishes displaying the data.

The scheduler will map block by block to available time intervals in the proper disk cluster. The assignment is done only if memory availability allows it. The following algorithm schedules the blocks of a request, starting by block number $BNumber$ and assuming that block $BNumber$ is displayed starting at $t_0$. The second argument, $t_0$, must be set to a very large number to represent infinity, for the case of the first block.

**Schedule($BNumber, t_0$)**

**While** $True$

Search for earliest time intervals in the cluster where the block $BNumber$ is, that satisfy memory constraints and can meet the deadline of start displaying the block at $t_0$.

**If** Search succeeded

**Then**

**If** $BNumber = 1$

Let $t_0 = Apply$ the algorithm in Section 4.1.2 to the first block.

**End If**

Update memory requirements for the time intervals when the block is scheduled and the time when memory is released

$t_0 = t_0 +$ display time of Block $BNumber$

**If** $BNumber$ is the last in the request

**Then** Return $True$

**Else If** Schedule($BNumber + 1, t_0$)

**Then** Return $True$

**End If**

**End If**

**End If**

**End While**

We now present the constraints for this model. We denote the average bandwidth requirement of a media type $A$ after compression by $\tilde{B}_{\text{display}}(A)$, where $B_{\text{display}}(A)$ is the bandwidth requirement of $A$ without compression. For example, given a media type $A$ with a bandwidth requirement of $B_{\text{display}}(A)$, if the average compression ratio for a media type $A$ is 50 to 1, we consider $\tilde{B}_{\text{display}}(A) = \frac{1}{50} \cdot B_{\text{display}}(A)$ as the average bandwidth requirement for objects of type $A$.

Constraints 12 and 13 remain unchanged. Constraints 10 and 11 are revised as follows:
(i) The block size is proportional to its average bandwidth requirement,
\[
\frac{n_1}{B_1} = \frac{n_2}{B_2} = \ldots = \frac{n_k}{B_k}
\]  

(ii) The memory requirements for the service of the queue of requests is always smaller than or equal to \(\text{size}(\text{mem})\). In particular, a block must fit in memory: For each \(i\), \(n_i \times \text{size}(\text{subobj}) \leq \text{size}(\text{mem})\).

The following new constraint assures the absence of hiccups:

Let \(o = o_1 \ldots o_m\) be an object such that for each \(k\), the piece \(o_k\) is read continuously in the time interval \([t^k_1, t^k_j]\). Let \(t_0\) be the start time to display \(o\). Let \(t_{\text{MaxLat}}\) be the maximum latency time incurred by activation and seek time. Then \(t_0 \geq t^k_1 + t_{\text{MaxLat}}\). And for all \(j > 1\), \(t_0 + t^o_{\text{display}} \geq t^k_j + t_{\text{MaxLat}}\) where \(t^o_{\text{display}}\) is the time to display \(o_1 \ldots o_{j-1}\).

5.1.2 Fixed Time Intervals and Fixed Block Sizes

The flexibility of the previous model could lead to excessive fragmentation of the time. Therefore, the objects may end up being partitioned into small pieces which implies excessive seek and latency times. Furthermore, its implementation could be computationally expensive. To overcome the problems just described, we establish a fixed size of the block and time intervals of fixed duration.

The scheduling under this approach is a special case of the scheduling procedure presented in Section 5.1.1. We now have blocks of fixed size, then the time to read a block is fixed. We map the time interval to the time to read a block and require to read the blocks within a time interval. Therefore, the fragmentation problem is no longer present and the search space of the scheduling algorithm is reduced, making the computation less expensive.

5.1.3 Configuration

To configure the system, we can apply the same method used for the non-compression case with different parameters. We now consider the average bandwidth requirements for the different media types after compression instead of the bandwidth disregarding compression. The other parameters remain the same.

For the case of Fixed Time Intervals and Fixed Block Sizes, a question that arises is how to define the block size if the configuration procedure gives a block size for each media type. There are several alternatives, it depends on what the priorities are. One extreme of the spectrum is to pick up the smallest block size arguing that memory is a very valuable resource, then it is
better to read the least to decrease the possibility of an object retaining memory for a long period. The other extreme is to select the largest block size to decrease the latency and seek time. One intermediate alternative is to compute an average block size based on the \textit{Heat} of the media types in consideration.

5.2 Split then Compress

When compression is applied to the blocks instead of the whole object, the block sizes become smaller, freeing disk bandwidth for other applications. For example, if the average ratio for image compression is 50 to 1, then the size of a block is reduced by a factor of 50 (in average). Which implies that in average the disk bandwidth utilization is $\frac{1}{50}$ of the utilization for the case when compression is not applied.

But, if the system is used exclusively for the display of objects, the released disk bandwidth can be used to service other objects. In that case, the model for non-compressed objects should be modified. We start by describing the differences with the model that does not consider compression. Then we give the revised constraints. And finally, we describe how to configure the system to handle compression.

Now we have the situation when the block sizes are not fixed but the display time of a block is still a cycle. In order to utilize the disk bandwidth as much as possible, we would like to read a block of another object (say B) shortly after a block of an object (say A) is read. But in the next cycle, the next block in A could be larger making the next block of B to be read at a different time interval. Therefore, the assumption of reading the blocks of an object in the same manner in all the cycles can not be made. The following example illustrates this issue.

\textbf{Example 5.1:} Let $A = a_1 \ldots a_m$, $B = b_1 \ldots b_k$ and $C = c_1 \ldots c_p$. A possible way of reading the objects is:

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c}

\hline
Cycle & Cycle & Cycle & Cycle & Cycle & Cycle & Cycle \\
\hline
$| a_1 | b_1 | c_1 | c_2 | a_2 | b_2 | \ldots$
\end{tabular}
\end{center}

Since the block sizes of objects A and B vary in the two cycles presented, the blocks must be read in different manner. For object C, the size is the same but the block $c_2$ had to be read at a different interval because the blocks of B and C took longer in the second cycle. $\square$

The irregularity of the distribution of a block within a cycle, requires additional memory. For this case, Lemma 4.6 is not longer true. But the memory requirement is still bounded by a fix number, namely $2 + n \times \text{size(subobject)}$. The worst case scenario is when a block is very small and
in one cycle is read at the beginning and the next cycle at the end. Therefore, the start of the object display must be done close to the end of the cycle. Which implies that the memory frame holding a block will be released at most 2 cycles later. Therefore, the upper bound of $2 + R + n \times size(subobject)$ is sufficient.

In summary, Constraints 10, 12 and 13 remain unchanged and Constraint 11 becomes:

$$2 + R + n \times size(subobject) \leq size(mem)$$

Notice that the block sizes $n_1, \ldots, n_k$ correspond to the sizes before compression.

To schedule requests we still consider cycles and schedule each block within a cycle. The only difference with the scheduler for non-compressed objects is that blocks of the same object are not read in exactly the same manner for all cycles and the block size is variable. The scheduler is very similar to the one for non-compressed objects. Basically, it keeps track of the requests scheduled for each cluster in each cycle and looks for a sequence of consecutive cycles that can read all the blocks of an object from the proper cluster.

The configuration procedure is very similar to the one for non-compressed objects. The only difference is the way the duration of the cycle is computed. For compressed objects,

$$n = \frac{size(mem)}{2 + \left\lfloor \frac{D}{r} \right\rfloor}$$

This approach is simpler than Compress then Split but it could eventually burn out disk bandwidth because of very small blocks after compression.

6 Conclusions and Future Work

In this paper we presented a new technique for supporting the continuous display of possibly compressed objects. This technique declusters the disk drives and strips the objects across the clusters. Also, it employs memory as an intermediate stage between the disk drives and the display stations. When displaying compressed objects, the bandwidth requirements become variable. Therefore, buffered I/O is fundamental to compensate the variable bandwidths.

We also presented the fundamental factors that impact the performance of the system and a configuration method that considers those factors. The configuration method introduced is aimed to achieve the optimal throughput for a given scheduling policy and a situation specified in the representative queue of requests. If the goal is to optimize throughput in a given situation (average,
worst, best case, etc.) the queue must represent the load in that situation. One interesting extension of this work, is the study of the behavior of a system with temporal and spatial constraints.

Considering the case when multimedia objects are instances of a type (class?) in an object oriented database, we have two approaches: sharing the buffer pool with the other types in the database or assigning a fraction of the memory to the multimedia types. We assumed the second approach, then memory frames used for the display of objects are not swapped out before their display. If the first approach is taken, the consideration of real time constraints in buffer pool management is an issue to be studied.

References


