A Design of Storage and Retrieval Schemes in a Multimedia Object Manager for Modeled Object and Efficient Multimedia Object Storage Strategies[1]

Kingsley C. Nwosu
IBM POWER Parallel Systems Division,
Neighborhood Rd., MS/992,
Kingston, NY 12401,
USA.

nwosuck@donald.aix.kingston.ibm.com.

Abstract

Due to the fact that the speeds of secondary storage devices have not improved relative to other computing technologies, a lot of efforts have been devoted to intelligent and efficient ways of storing data of storing data to maximixe the bandwidth of a computing system. Many of those techniques have proven useful; however, with the advent of multimedia and its enormous real-time and size requirements, the problems of data allocation and retrieval have become even more crucial. In [1][2], we describe some designs of efficient multimedia object storage strategies that strive to allocate multimedia objects with the objective of satisfying their real-time retrieval requirements. We utilize some fragmentation strategies, a cost function, and bipartite graph model to allocate multimedia data efficiently. In the development of the storage allocation strategies, some assumptions were made about the functionalities and capabilities of the controlling Multimedia Object Manager. Here, we describe the design of the storage and retrieval schemes for a multimedia object Manager (MOM) that controls our allocation techniques. We describe the multimedia object model, generation and aggregation of the Storage Elements (SEs), determination of SEs spanned by an I/O request, clusterization and parallelization of requests, storage and retrieval structures, and extension and reduction of objects.

Keyword: efficient storage, multimedia object modelling, object decomposition, parallel I/O, retrieval schemes, storage schemes.

1 Introduction

The term **multimedia** means different things to different people depending on their operating environment. To some people it simply means the integration of text and graphics on a stand-alone system like a personal computer. To others it means the integration of text,

[1] *Published in Proc. of 2nd Intern. Conf. on Info. Syst. & Management of Data (CISMOD), New Delhi, India, Oct. 3-5, 1993*

graphics, and audio. Irrespective of one's view of multimedia, it is a combination of two or more of the following: motion and still video, special effects, synthetic video, graphics, text, voice, audio, and images. The management of multimedia data is one of the most crucial features of multimedia information processing. Although the cost of the hardware required for the capture, storage, and presentation of multimedia data is decreasing every year, the softwares for effectively and efficiently managing such data are lacking. It is a *sine quo non* that intelligent and efficient multimedia data storage and management techniques be developed for multimedia information processing. Processor speed, memory speed, and memory size have grown exponentially over the past few years [3][4], however, disk speeds have improved at a far slower rate. Consequently, the speed of the disk rather than the speed of the CPU's is the limiting factor in many applications. The bandwidth of the secondary storage devices has not improved relative to other technological advances.

Conventional allocation techniques (such as data stripping/de-clustering [5][6][7] and data contiguity/clustering [8][9][10]) are developed mainly for text and numeric files, which although can be different in sizes, are more or less on the same order. Unfortunately, when applied to multimedia applications, the conventional techniques are inadequate and inefficient. For real-time information retrieval and presentation, it is imperative that data, for a given medium, be retrievable at some given rate. The rates for some media are very high for current storage devices. The most conspicuous of these is in the area of digital video. For example, the video data object based on the **NTSC** standard requires that video data be retrievable at a rate of 45 MB/s. However, the peak speed of a magnetic disk drive is about 10MB/s and CD-ROMs operate at 1.2 MB/sec. Due to this apparent limitation on the secondary storage devices, it is necessary to store data intelligently in order to obtain the expected retrieval speed. Consequently, the way multimedia objects are stored and the capabilities and efficiency of the object manager play a large role in determining the realization of efficient and reliable multimedia information processing system.

In [1][2], we describe the multimedia allocation problem and the reasons that conventional approaches have proven inadequate and insufficient. We presented a multimedia object model, the fragmentation strategies, and the efficient allocation process for parallel access storage devices and configuration as depicted in Figure 1. In the design of those allocation strategies, we assumed the existence of a Multimedia Object Manager (MOM). A MOM is a subsystem of a Multimedia Server that is responsible for the storage and management of different data allocations and retrievals. A Multimedia Server does not only manage complex objects, but also must be capable of integrating various information units for complex object composition. It must also provide capabilities for enforcing consistency, protecting objects, ensuring synchronization, etc.

Figure 1: Parallel Access Storage Device Architecture.

In this paper, we present a design of efficient multimedia object storage and retrieval schemes based on our allocation strategies. These schemes strive to achieve data availability at some specified rates for real-time multimedia information presentations.

The rest of the paper is organized as follows: Section 2 describes the multimedia object model and decomposition techniques, and Section 3 discusses the generation and assignment of SEs to the allocation units. In Section 4, we show how to determine the SEs spanned by a retrieval request. Section 5 presents the clusterization and parallelization of a retrieval request. In section 6 we show the storage and retrieval structures for multimedia object storage and retrieval. Section 7 describes how DEs can grow or shrink while Section 8 discusses, in general, the necessary and sufficient steps for a retrieval process.

2 Multimedia Object Data Model

The multimedia object model comprises a number of tree-structured collection of objects called *complex multimedia objects* that form a unit called a *composite object.* Each multimedia object, that is a leaf node, in a composite object is a media type whose storable data is referred to as a *Data Element* (DE). Figure 2 shows a composite object $o1$ that consists of 3 complex multimedia objects $\{o_2, o_3, o_6\}$ while the multimedia objects $\{o_4, o_5, o_8, o_9, o_7\}$ are the DEs. The DEs are further classified into three different classes, namely, *class-one, class-two* and *class-three* based on their necessity to meet an *expected retrieval rate*, a *degree of parallelism* or no I/O real-time requirement. A class-one DE requires that some minimum amount of its data be retrievable per unit time. This amount of data is referred to as a DE's *expected retrieval rate.* A class-three DE has a degree of parallelism needed to enhance its computational efficiency for multi or vector processing.

Figure 3: Examples of class-one, class-two, and class-three DEs in a composite object.

that represents a storage device's bandwidth.[1] An AU stored in a storage device consists of a number of SEs each of size equal to the storage device's bandwidth. These SEs are arranged in the corresponding AUs with the aim of ensuring parallel accesses to media data by accessing different AUs to meet the real-time requirements. Each composite multimedia object, after decomposition, comprises a number of AUs which are allocated individually on separate storage devices. For example, using the DEs in Figure 3 and assuming that the sizes of DE1, DE2, DE3, and DE4 are 320KB, 50KB, 100KB, and 500KB, respectively, therefore, in a homogeneous storage device configuration (where the bandwidths of the storage devices

[1]number of bytes of data that is retrievable from a storage device per unit time

are the same), the DEs are decomposable into 7 SEs, $\{se1, \ldots, se7\}$, 1 SE, $\{se1\}$, 2 SEs, $\{se1, se2\}$, and 10 SEs, $\{se1, \ldots, se10\}$, respectively, and the AUs are constructed as shown in Figure 4. We assume that the bandwidth of the storage devices is 50KB per unit time. The numbers below the SE numbers in the boxes represent the *physical addresses* of each SE within its AU. We assume that each AU's physical address starts from zero. For example, $se3$ in DE4 spans the physical addresses from 100 to 149. In a homogeneous storage device configuration, the size of all the SEs of all the AUs of a multimedia object is the same, thereby, making the offsets between different SEs in different AUs uniform. This uniformity does not exist for heterogeneous storage device configurations. For heterogeneous storage device configuration, a number of differences in bandwidths. For example, assuming that we

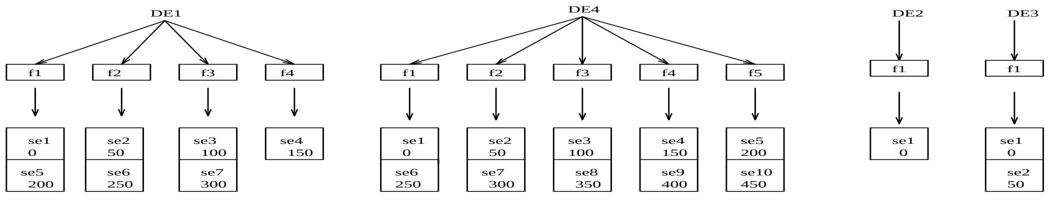

Figure 4: Homogeneous devices: sample construction of AUs of a composite object.

have storage devices whose bandwidths are either 50KB or 100KB per unit time, therefore, using the Figure 3 example again, we can construct 5 SEs for DE1, 7 SEs for DE4, 1 SE for DE2 and 1 SE for DE3 as shown in Figure 5. These are one possible generation of AUs; there are other ones. In [1][2], we present the allocation process that efficiently maps each

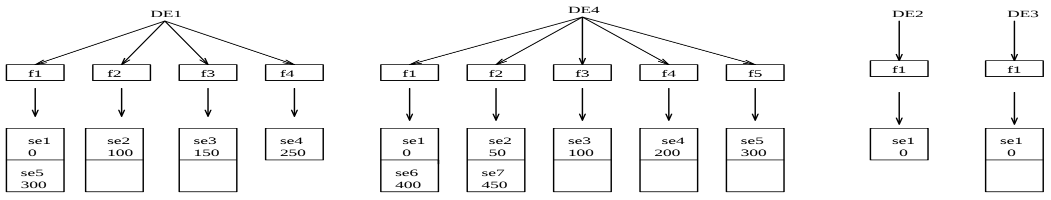

Figure 5: Heterogeneous devices: sample construction of AUs of a composite object.

AU of a complex multimedia object to a storage device. During that allocation process, we determine, for each AU, its *size, location, unit*, etc. The *size* of an AU is the total number of bytes of data that it contains; the *unit* is the size of each SE in the AU; while the *location* identifies (by device identification) the storage device to which it is allocated. During allocation, each DE stores its *decomposition factor* which comprises the number of AUs generated by the DE and the unit of the AUs.

3 Generation and Assignment of SEs to AUs

As we stated earlier, a DE comprises a number of AUs each consisting of a number of SEs. During the determination of the efficient allocation of the AUs of a composite multimedia object to storage devices, we constructed the AUs with the primary objective of achieving the expected retrieval rate or degree of parallelism. Furthermore, we only determined the size of each AU from its unit and each AU is a logical representation of some data. Each AU does not indicate which physical data from the DE that it represents. In order to store actual data via the AUs we must determine the physical constituents of the AUs with respect to the SEs. Given an index of a SE of a DE, we want to be able to determine the AU (a_k)to which it belongs. In order to accomplish this, we must have a way of uniquely identifying each SE in a DE. However, firstly, we have to compute the number of SEs in a DE. As a prelude to computing the number of SEs, we solve an integer linear programming problem. We group related storage devices together based on their bandwidths into $\wp^1, \wp^2, \ldots, \wp^\mu$, where μ is the numberof different bandwidths in the system and the bandwidth of each storage device in $\wp^i$ is $BW(\wp^i)$. Subsequently, given that the expected retrieval rate of a DE is r and its size is s, we solve the problem:

$$y_1 BW(\wp^1) + y_2 BW(\wp^2) + \ldots + y_n BW(\wp^n) \geq r,$$
$$y_1 \leq y_2, y_2 \leq y_3, \ldots, y_{n-1} \leq y_n, y_n > 0.$$

Also, $BW(\wp^1) \geq BW(\wp^2) \geq \ldots \geq BW(\wp^n)$. The solution from the problem above produces the number of storage devices of each group of devices that is needed to achieve the real-time requirement and also the total number of AUs. The same problem is solved by substituting r for s to determine the number of SEs of each bandwidth needed. The computation of the number of AUs and SEs and their acceptability are detailed in [1][2]. We then compute the total number of SEs in a DE as

$$\delta = \textstyle\sum_{i=1}^{n} y_i.$$

Having computed the number of SEs in a DE, we uniquely identify each SE. We denote the fact that the qth SE belongs to a_k as $\Join_k^q$. Given that a DE comprises δ SEs, we generate δ number of SEs indices and using the constituents of δ, as above, we initially know that

$$\forall_{i=1,\gamma}\ q_i = i \text{ and } \Join_i^{q_i}.$$

Subsequently,

$$\forall_{j=\gamma+1,\delta} \text{ if } j \ mod \ \gamma = q_i \ (1 \leq i \leq \gamma) \text{ then } \Join_i^j.$$

At this time, we have generated all the SEs of a DE, indexed and assigned them to the AUs, now, we have to determine the physical address of each SE in an AU. Henceforth, we denote the ith SE as se_i, number of AUs generated by a DE as γ, and unit of a_i as τ_i. Assuming that $ADDR(se_i)$ is the physical address of se_i, then

$$ADDR(se_i) = \begin{cases} \sum_{k=1}^{\gamma} \tau_k w + (\sum_{j=1}^{n'} \tau_j) & \text{if } w > 0 \\ \sum_{k=1}^{n'} \tau_k & \text{otherwise} \end{cases}$$

where

τ_k, $(1 \leq k \leq \gamma)$ is the size of each SE in a_k,
$w = \lfloor \frac{i-1}{\gamma} \rfloor$,
$n' = (i-1)\ mod\ \gamma$.

For example, from Figure 5, where $\tau_1 = 50,\ \tau_2 = 50,\ \tau_3 = 100,\ \tau_4 = 100$, and $\tau_5 = 100$, for $se6$ of DE4, $w = 1$, and $n' = 0$. Therefore,

$$ADDR(se6) = [50.1 + 50.1 + 50.1 + 100.1 + 100.1 + 100.1] = 400.$$

Similarly, for $se7$ of DE4,

$$ADDR(se7) = ADDR(se6) + 50 = 450.$$

Another property of a SE that is important for satisfying an I/O request is the physical location of a SE in the storage device to which the corresponding AU is stored. After the allocation process, each AU carries with it its starting location in the storage device to which it is allocated. If ν_k is the starting location of a_k in a storage device and $LOC(se_i)$ is the physical location of se_i, then

$$LOC(se_i) = \begin{cases} \tau_h w + \nu & \text{if } h \neq 0 \\ \tau_n w + \nu & \text{otherwise} \end{cases}$$

where

$w = \lfloor \frac{i-1}{\gamma} \rfloor$, and
$h = i\ mod\ \gamma$.

Using the set up of the example above, assuming that $\nu_1 = 100$ and $\nu_2 = 50$, for $se6$ and $se7$ of DE4, $w = 1$ and $h = 1,\ h = 2$, respectively, where

$$LOC(se6) = 100 + 50 = 150,$$
and
$$LOC(se7) = 50 + 50 = 100.$$

4 Determining the SEs of a Retrieval Request

A typical retrieval request, in most computing systems, consists of a physical address and a size. The physical address specifies the location in the DE to start retrieving data while the size specifies the amount of data in bytes to retrieve from the starting address. As a result, any reliable scheme to facilitate the retrieval of the multimedia object data must have the capability of determining which AUs contain the starting request address and which SEs are spanned by the request size. In general, given a retrieval request of size θ that starts at ϑ, the SEs spanned by the request are

$$\{se_{x_1}, se_{x_2}, \ldots, se_{x_z}\}$$

where

$$x_1 = \begin{cases} \gamma w + w^{'} + 1 & \text{if } w > 0 \\ w^{'} + 1 & \text{otherwise} \end{cases}$$

and

$$h = \textstyle\sum_{i=1}^{\gamma} \tau_i,$$
$$w = \lfloor \tfrac{\vartheta}{h} \rfloor,$$
$$0 \leq w^{'} \leq \gamma - 1 \text{ and } \textstyle\sum_{j=1}^{w^{'}} \tau_j \leq (\vartheta - wh) \leq \sum_{j=1}^{w^{'}+1} \tau_j.$$

Henceforth, we refer the value represented by h as the *storage length data* of a DE. The index, x_z, of the last SE spanned by the I/O is computed similarly by substituting ϑ by $(\vartheta + \theta)$. For example, using Figure 5, let $\vartheta = 55$ and $\theta = 200$, therefore, for x_1,

$$h = 400,$$
$$w = 0,$$
$$0 \leq 1 \leq 4 \text{ and } 50 \leq 55 \leq 100.$$

Therefore, $x_1 = 2$. For x_z, $\vartheta = 255$, and

$$w = 0,$$
$$0 \leq 3 \leq 4 \text{ and } 200 \leq 255 \leq 300.$$

Therefore, $x_z = 4$. Consequently, the I/O request spans $\{se_2, \ldots, se_4\}$.

5 Clusterization and Parallelization of a Retrieval Request

Since the *storage length*[1] of a DE represents the number of storage devices spanned by the SEs of a DE to achieve the real-time requirements, under normal circumstance, any retrieval request to the DE can be performed by a number of parallel requests limited by the storage length of the DE. One of the primary goals of our allocation strategy is to provide the ability to initiate parallel requests to achieve some real-time requirements. Therefore, having determined the SEs spanned by a retrieval request, we group the SEs such that a group comprises SEs that are allocated to the same storage device and their locations are monotonically increasing by the storage length data of the corresponding DE. The locations of an arrangement of a number of SEs can only be monotonically increasing with respect to the storage length data if and only if those SEs are contiguously allocated. However, an extension of a DE (as described below) may violate this contiguity rule. Under that situation, a request may necessitate a number of parallel requests that is greater than the storage length of the DE. A parallel request must span some SEs whose locations are monotonically increasing with respect to the DE's storage length data and the SEs must belong to the same storage device. As a result, we satisfy a retrieval request by initiating a number of parallel requests equal to the groups formed by the SEs spanned by the request. For example, a request to DE4 in Figure 5 starting from the physical address 150 and size of 270 bytes spans $se3$ to $se6$ necessitating 4 parallel I/Os for f_3, f_4, f_5, and f_1.

6 Storage and Retrieval Schemes

One of the objectives of the design of the storage and retrieval schemes for the multimedia object model described is to maintain compatibility and implementability with UNIX[2]-oriented [11][12] [13][14] computing systems. UNIX-based systems operate with hierarchical stream-oriented file systems. A file system is the most conspicuous component of an operating system that is responsible for storing, retrieving, naming, protecting, organizing, etc., most of a computing system's data and applicable resources. The primary purpose of a file system is to store and manage data effectively and reliably. The storage and retrieval schemes presented here are also extensible to incorporate object-oriented model of a file system. The storage and retrieval schemes for our model of the MOM is designed with a goal to fostering both direct and indirect manipulation of the multimedia data through the file system and MOM. We want to be able to access the multimedia data via the file system for read-only operations and reserve the rights for read/write operations to the MOM. The details and specifics of the interactabilities between the file system, the MOM and the DEs will be presented in a follow up to this paper.

6.1 Storage and Retrieval Structures

The Multimedia Object Manager that controls the storage and retrieval process for the multimedia object model described here consists of an **AU Hash Table** (*AUHT*) and a

[2]UNIX is a registered trade mark of AT&T

number of **SE nodes** (*senodes*) for each DE mapped to some storage devices.

Definition: An *AU Hash Table* is a segmented table that contains an *AUHT header*, a number of pseudo-AU segments (*pausegs*) and, possibly, some extension segments (*extsegs*) for each DE allocated in the system. The *AUHT header* contains such information as the decomposition factor, unique identification for the DE, current number of *pauseg* and *extseg* entries, etc. Each *pauseg* contains a pseudo-AU segment identification (*psegid*), a starting address (*saddr*), an ending address (*eaddr*), a segment modifier (*smod*), a SE modifier (*semod*), a pointer to a list of *senodes* (*selist*), a number of *senodes* (*nses*), and some reserved space (*segres*). The *psegid* is a positive integer value that indexes the *pausegs* and also denotes the relative time of the creation of a *pauseg*. The *saddr* is the lowest physical address of the SEs pointed to by *selist*. The *eaddr* is the highest byte address in the SE that has the highest physical address in the SEs pointed to by *selist*, i.e., if se_z is the SE with the highest physical address, then the *eaddr* is the sum of the physical address of se_z and the size of se_z minus 1. The *smod* is a signed integer value that is used to normalize the *saddr* and *eaddr* of a *pauseg* and the physical addresses of the SEs pointed to by *selist*. The *semod* is a signed integer value that is used to normalize the unique identifications of the SEs pointed to by *selist*. The *selist* points to the SEs spanned by *saddr* and *eaddr*. The *nses* indicates the number of *senodes* that are associated with a given *pauseg*. The *segres* is some reserved space that could be used by the MOM to store some special information about a *pauseg*. In case an AUHT block is exhausted, each AUHT contains a *nauht* field for additional information.

Definition: An *extseg* is a fixed size data entity that is exactly the same size and contains the same fields as a *pauseg*. We usually associate a *pauseg* with an allocation that does not violate the real-time requirements and the sequential nature of DE. An *extseg* is used to maintain the allocation and real-time requirements of DEs after extension and reduction (as explained below). Each field of an *extseg* carries the same information as in a *pauseg*.

Definition: An *senode* is a fixed size data entity that comprises a SE identification (*seid*), a SE physical address (*sepaddr*), a SE size (*sesize*), a storage device identification (*sdid*), a SE location (*seloc*), and some reserved space (*seres*).

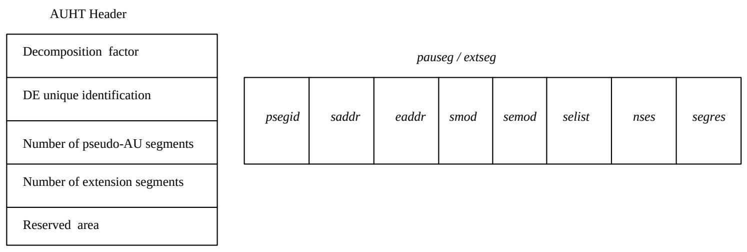

Figure 6: Layout of an AUHT Header and a *pauseg* or an *extseg*.

The *seid* is the unique SE number assigned to a SE during the generation and assignment of SEs to AUs (See Section 3). The *sepaddr* is the SE physical address that represents the physical starting address of the SE in the DE. The *sesize* is the actual size of the SE which is also the number of bytes that are sequentially spanned by the SE. The *seloc* is the byte address of the block where the SE is stored in a storage device. The *sdid* is the unique identification of the storage device on which an *senode* is stored. For all practical purposes, the *sdid* contains all the pertinent information needed to locate a storage device whether it is local or remote and for manipulation by the file system. The *seres* is some reserved space that could be used to store special information by the MOM. It could also be used to store some small amount of data of a SE, especially after extension, depending on the size of *seres* and the data. Figure 7 shows the layout of an AU Hash Table and an *senode* while Figure 8 shows the AUHT and *senodes* of DE4 from Figure 5.

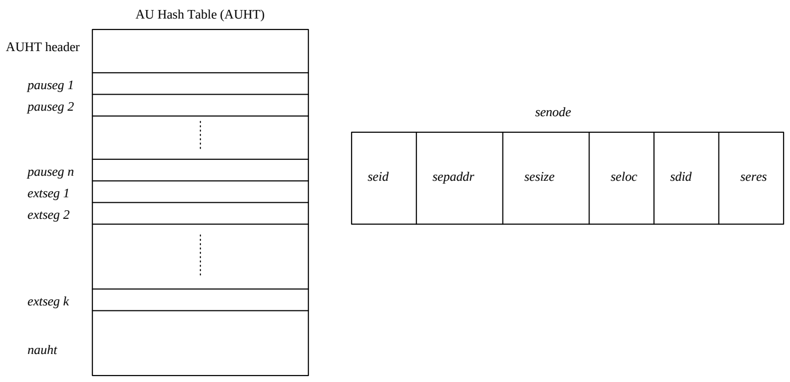

Figure 7: Layout of an AU Hash Table and an *senode*.

7 SE Extension and Reduction Process

Two of the most costly problems in maintaining data are the effects of data extension and reduction. It is relatively easier to decompose and store data after the initial creation. However, when a fully allocated SE of an already stored DE is extended, the data allocation must be performed reliably and correctly so that the natural sequentiality of the data and the allocation requirements are not destroyed. It is obvious that when a SE of a currently allocated DE is extended, the new data upsets the current physical addresses and, thereby, renders the storage of some data, with respect to the data set that must be retrieved in parallel, invalid. For example, in Figure 1, assuming that *senode 2* of *DE*4 is extended with 30 bytes of data, the new or extraneous data, following our allocation policy, should be in *senode 3*, and the last 30 bytes of data in *senode 3* should be in *senode 4*, and so on. Obviously, this kind of change has upset our allocation objective.

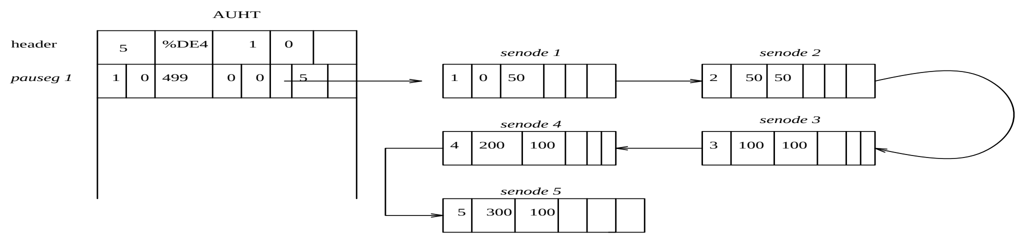

Figure 8: An example of an AUHT and *senode* entries.

In conventional Unix-based and other file systems, this kind of problem is negligible since files are not allocated with the objective of ensuring some parallel retrievability. A common but costly approach to handle the problem is to always re-allocate a DE after extension.

We can confidently and consistently allocate some new data resulting from an extension if the size of the new data is equal to its DE's storage length data. In that case, the new SEs can be allocated to the storage devices holding the DE according to the allocation policies since the SEs span the DE's storage length. For example, if we extended *senode 2* in Figure 8 by 400 bytes, we can confidently and fully allocate the new data across the DE's storage length devices, thereby, uniformly changing some of its physical addresses. As a result, we store new or extra data resulting from extensions on the DE's storage devices only when the data forms a number of SEs equal to the storage length of the DE. In order to accomplish this, it becomes imperative that we must be able to maintain and coalesce data resulting from extension until they form the required SEs. We use the *extsegs* to maintain the list of contiguous data from extension that have not formed the required allocation unit. Each *selist* of an *extseg* points to a list of *senodes* whose cumulative size is less than the DE's storage length data and whose physical addresses are monotonically increasing. Until extension data forms an allocatable unit, it is stored in some reserved spaces in the storage devices that hold the DE.

As a result of extending a fully allocated SE, the *senodes* of a newly created SE is updated with information from the *senode* of the original SE and certain fields of the existing *pausegs* are modified. The *sepaddr* of the new SE is computed from the *sepaddr* and *sesize* of the original SE. If the new *senode* is not a logical part of an existing *extseg*, then a new *extseg* is created and its fields' information determined from the current *pauseg* or *extseg*. It is also assigned the next available index number from the *psegid* of the original *senode*. All the *senodes* following the *senode* that was extended are detached from their current *pauseg* and a new *pauseg* entry is created for them, updated and indexed accordingly. Furthermore, every *smod* value for every segment entry whose *saddr* is greater than or equal to the *saddr* of the new *pausegs* or *extsegs* is modified with the number of bytes by which the DE was extended. Every *semod* of every *pauseg* and *extseg* whose *saddr* is greater than or equal to the *saddr* of the new *pauseg* or *extseg* is modified with the number of SEs by which the DE was extended. The *semod* and *smod* of the newly created segment entry are not affected.

If the SE extended is not fully allocated, then one needs only to update the *smod*'s of the applicable segment entries. Furthermore, an extension on the last *senode* of a DE does not require that an *extseg* be created; we simply use the index of the new SE to determine which storage device to store it. For example, for DE4 of Figure 5, assuming that $f_i \perp$[2] S_i, $(1 \leq i \leq 5)$, and DE4 has been extended by 300 bytes. This extension produces $se8, se9$, and $se10$ which, using the process described in Section 3 and the storage length of DE4, shows that $\bowtie_3^8, \bowtie_4^9$, and $\bowtie_5^{10}$.

When we need to update an *smod* of an entry after an extension, we add the size of the extension to the current value of the *smod*. Furthermore, if some new SEs were created, we add the number of SEs created to the current value of the *semod* of each of the segment entry whose *saddr* is greater than the *saddr* of the extended segment entry. Figure 9 shows the new values for the AUHT and *senodes* from Figure 8 after extending *senode 2* by 70 bytes and *senode 4* by 130 bytes. Figure 10 shows the results of extending *extseg 2* of Figure 9 with some data such that the storage length data is achieved. In this case, *extseg 2* was extended with 270 bytes.

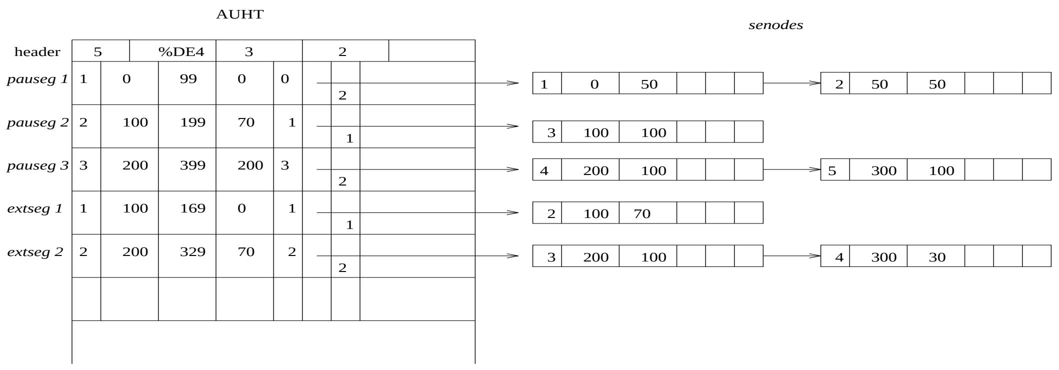

Figure 9: An example of extending *senodes*.

Having achieved the storage length data from the extension, the *extseg 2* data are moved from the extension area to applicable storage devices. During extension, the size of an *senode* is determined by applying its expected *seid* to the storage length to determine the storage device it should be stored. The following arrangement then results: $\bowtie_4^5, \bowtie_5^6, \bowtie_1^7, \bowtie_2^8, \bowtie_3^9, \bowtie_4^{10}$, and $\bowtie_5^{11}$. In the case of removing an allocated SE, we run into the problem of holes which can only be closed by coalescing. Since the DE has been successfully allocated, we can coalesce after reduction without running the risk of insufficient space or destruction of the data arrangement for real-time requirements.

[2] $a \perp b \Rightarrow$ AU a is stored in storage device b

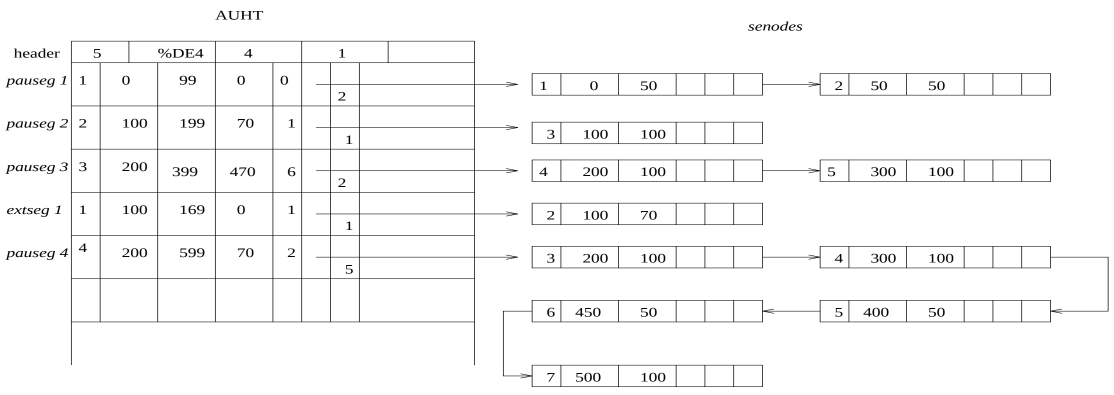

Figure 10: A sample extension that forms a storage length data.

The coalescing can be achieved by direct copying of SEs or bit ORing. The affected *pausegs* and *senodes* are modified accordingly by performing reverse operations to extension. Figure 11 shows the results of removing *senode 4* in Figure 10.

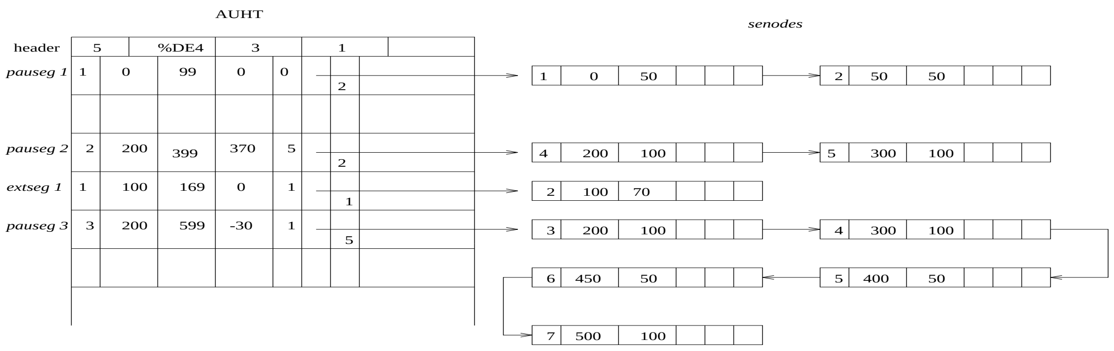

Figure 11: A sample removal/reduction of an *senode*.

Having shown some of the examples of data extension and reduction updates in our storage and retrieval schemes, we now state some of the primary operations in those processes. Let Π^p be the current number of *pausegs* and Π^e the current number of *extsegs* in a DE, $SEG_i(attr)$ the value of attribute *attr* for the *pauseg* or *extseg* entry whose *psegid* is i and $SENODE_k^i(attr)$ the value of attribute *attr* for the *senode* in SEG_i whose position is k in *selist*. If $SENODE_l^i$ is fully allocated and has been extended by s' SEs of total size s'', then one of the following cases and major operations arises:

1. l is equal to $SEG_i(nses)$ and SEG_i is an *extseg*
   (a) The sum of $SEG_i(nses)$ and s' is less than the DE's storage length:
      Attach the new *senodes* to the *senodes* of $SEG_i(selist)$,
      $\forall\ SEG_k \in pauseg \vee extseg\ (i \neq k)$, if $SEG_k(saddr) \geq SEG_i(saddr)$
      then $SEG_k(smod) = SEG_k(smod) + s''$,

```
            $SEG_k(semod) = SEG_k(semod) + s^{'}$,
        $SEG_i(nses) = SEG_i(nses) + s^{'}$.
  (b) The sum of $SEG_i(nses)$ and $s^{'}$ is greater than or equal to the storage length:
        Attach the new *senodes* to the *senodes* of $SEG_i(selist)$,
        $k^{'} = SEG_i(nses) + s^{'}$,
        $SEG_i(eaddr) = SEG_i(eaddr)$+size of $\{SENODE^i_{SEG_i(nses)+1} \ldots SENODE^i_{\gamma}\}$,
        $SEG_i(nses) = \gamma$,
        Change $SEG_i$ to a *pauseg*.
        If $k^{'} > \gamma$,
        then $\Pi^e = \Pi^e + 1$,
            Create *extseg* $SEG_{\Pi^e}$ from $SEG_i$,
            $SEG_{\Pi^e}(saddr) = SEG_i(eaddr) + 1$,
            $SEG_{\Pi^e}(selist) \rightarrow SENODE^i_{\gamma+1}$,
            $SEG_{\Pi^e}(eaddr)$ = size of $\{SENODE^i_{\gamma+1} \ldots SENODE^i_{k^{'}}\} - 1$,
            $SEG_{\Pi^e}(nses) = k^{'} - \gamma$,
        $\forall\ SEG_k \in pauseg \vee extseg\ (i \neq k, i \neq \Pi^e)$, if $SEG_k(saddr) \geq SEG_i(saddr)$
        then $SEG_k(smod) = SEG_k(smod) + s^{''}$,
            $SEG_k(semod) = SEG_k(semod) + s^{'}$,
2. Otherwise
        $\Pi^e = \Pi^e + 1$,
        Create *pauseg* $SEG_{\Pi^p+1}$ (if $l \neq SEG_i(nses)$) and *extseg* $SEG_{\Pi^e}$ from $SEG_i$,
        $SEG_i(eaddr)$ = size of $\{SENODE^i_1 \ldots SENODE^i_l\} - 1$,
        If $\exists\ SEG_{\Pi^p+1}$
        then $\Pi^p = \Pi^p + 1$,
            $SEG_{\Pi^p}(saddr) = SEG_i(eaddr) + 1$,
            $SEG_{\Pi^p}(eaddr)$ =size of $\{SENODE^i_{l+1} \ldots SENODE^i_{SEG_i(nses)}\} - 1$,
            $SEG_{\Pi^p}(nses) = SEG_i(nses) - l$,
            $SEG_{\Pi^p}(selist) \rightarrow SENODE^i_{l+1}$,
        $SEG_i(nses) = l$,
        $SEG_{\Pi^e}(saddr) = SEG_i(eaddr) + 1$,
        $SEG_{\Pi^e}(eaddr) = SEG_{\Pi^e}(saddr) + s^{''} - 1$,
        $SEG_{\Pi^e}(nses) = s^{'}$,
        $\forall\ SEG_k \in pauseg \vee extseg\ (i \neq k, i \neq \Pi^e)$, if $SEG_k(saddr) \geq SEG_i(saddr)$
        then $SEG_k(smod) = SEG_k(smod) + s^{''}$,
            $SEG_k(semod) = SEG_k(semod) + s^{'}$,
```

In the case of removal/reduction, we apply similar steps by de-attaching *senodes* instead of attaching them, performing subtraction instead of addition, and modifying the locations and sizes of *senodes* appropriately.

8 General Object Retrieval Process

Although we have described the techniques and operations necessary and sufficient for a DE extension or reduction, however, the expected bulk of the activities in our system are retrieval requests. Here, we enumerate the primary steps necessary for a successful completion of a retrieval request. A retrieval request comprises a DE *name*, an *offset*, and a *size*. The *offset* is the physical position within *name* where a retrieval should start while *size* is the total amount of data to retrieve. Therefore, given the above retrieval information, the following steps are necessary to accomplish the request:

1. Obtain the AUHT of *name*,
2. Find the *pauseg/pausegs* and/or *extseg/extsegs* that cover the *offset* and *size*,
3. Collect all the *senodes* that span the *offset* and *size* from information obtained in step 2,
4. Using the *sdid* of the *senodes* collected in step 3, group related *senodes* in increasing order of their locations with respect to their sizes,
5. Issue concurrent requests for all the groups formed.

9 Simulation and Proto-typing

The simulation results of the allocation strategies mentioned here are presented and analyzed in [1][2]. The simulations involved the generation of DEs of composite objects and storage devices with the necessary attributes. The decompositions and the allocation parameters were applied with the storage techniques which produced, in addition to satisfying the real-time requirements, balanced distribution of objects in the storage devices. Figure 12 shows an example result of such distribution where the cost values of allocated objects within a group of homogeneous storage devices are relatively uniform. The *final cumulative traffic requirement* is the weight cost value of the DEs allocated to a storage device. A proof of concept model has been developed and implemented for the storage schemes described here for distributed multimedia environments.

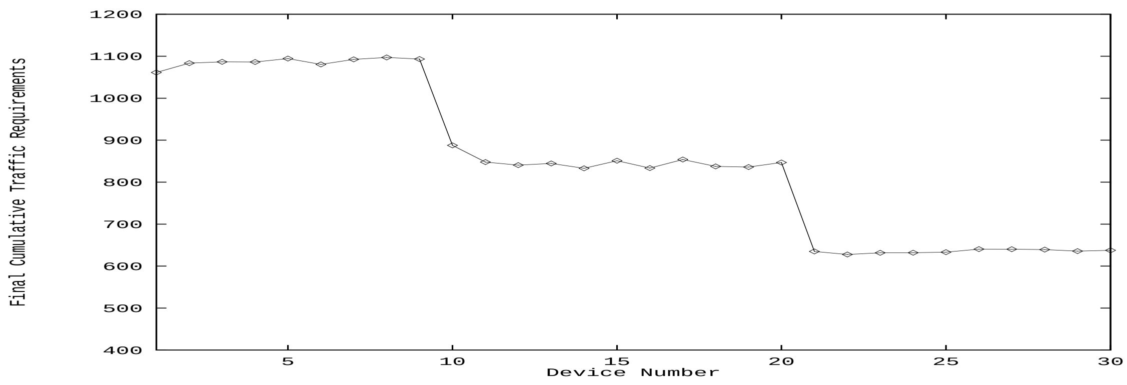

Figure 12: A sample data distribution with our allocation technique.

10 Conclusions

In this paper we have described the design of the storage and retrieval schemes in a multimedia object manager with respect to the multimedia object model and storage allocation strategies described in [1][2]. Although there are numerous issues in a multimedia object manager, here we focus on the storage and retrieval schemes. We have presented descriptions of the multimedia object model, the generation and assignments of the SEs to AUs, determination of the SEs spanned by a retrieval request, clusterization and parallelization of SEs spanned by a retrieval request, storage and retrieval structures, extension and reduction of SEs, and the necessary updates to the storage structures. We developed our schemes with a goal of inter-operatability with conventional UNIX file systems. Interactions via the file systems have read-only access while the MOM has the capability to extend or reduce a DE.

References

[1] **C. Y. Roger Chen, Kingsley C. Nwosu, P. Bruce Berra,** Modeling and Storage Allocation Strategies for Homogeneous Parallel Access Storage Devices in Real-Time Multimedia Information Processing, *to appear in the Proc. IEEE 5th International Conf. on Computing and Information* (ICCI), Sudbury, Ontario, Canada, May 1993.

[2] **C.Y. Roger Chen, Kingsley C. Nwosu, P. Bruce Berra,** Multimedia Object Modeling and Storage Allocation Strategies for Heterogeneous Parallel Access Storage Devices in Real-Time Multimedia Computing Systems, *to appear in the Proc. 17th International Computer Software and Applications Conf.* (COMPSAC), Phoenix, Arizona, November, 1993.

[3] **Bell, C. G.**, The mini and macro industries, *IEEE Computer, Vol. 17, No. 10*, 1984, pp. 14-30.

[4] **W. Myers**, The Competitiveness of U.S.A. Disk Industry, *IEEE Computer*, Vol. 19, No. 11, 1986, pp. 85-90.

[5] **Kim, M. Y.**, Synchronized Disk Interleaving, *IEEE Trans. on Computers, Vol. C-35, Vol. 11*, 11/86, pp. 978-988.

[6] **Livny, M., Khoshafian, S., and Boral, H**, Multi-Disk Management Algorithms, *Proc. 1987 ACM SIGMETRICS Conf. on Measurement and Modeling of Comp. Syst.*, pp. 69-77.

[7] **Chen, P. and Patterson, D.**, Maximizing Performance in a Striped Disk Array, *Proc. 1990 ACM SIGARCH 17th Intern. Symp. on Comp. Arch.*, Seattle, WA, May 1990, pp. 322-331.

[8] **J. Zupan**, Clustering of Large Data Sets, *Technometrics*, Vol. 29, Nov., 1987, pp. 497.

[9] **A.D. Bell, F.J McErlean, P.M. Stewart**, Clustering Related Rules in Databases, *The Computer Journal*, Vol. 31, June 1988, pp. 253-257.

[10] **J.S. Deogun, V.V. Raghava, T.K.W. Tsou**, Organization of Clustered Files for Consecutive Retrieval, *ACM Trans. on Database Systems*, Vol. 9, December 1984, pp. 646-671.

[11] **Maurice J. Bach**, The Design of the UNIX Operating System, *Prentice Hall*, Englewood Cliffs, New Jersey, 1986.

[12] **S. R. Bourne**, The UNIX Sytsem, *Addision-Wesley*, Reading, MA, 1983.

[13] **M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry**, A Fast File System for UNIX, *ACM Trans. Comput. Syst.*, 2(3), Aug. 1984, pp. 181-197.

[14] **K. Thompson**, Unix Implementation, *Bell System Technical Journal*, 57(6), July 1978, pp. 1931-1946.