Object Placement in Parallel Hypermedia Systems*

Shahram Ghandeharizadeh Luis Ramos Zubair Asad
Waheed Qureshi

Computer Science Department
University of Southern California

Abstract

During the past few years, hypermedia systems have emerged as an essential component of many application domains ranging from software engineering to library information systems. This is primarily due to the capability of these systems to manage and represent irregularly structured information, and to provide a user-friendly interface for information retrieval by providing a browsing capability. However, most stand-alone implementation of these systems (based on a workstation) cannot support a "real-time" display of audio and video objects. This is due to the low I/O bandwidth of the current disk technology and the large size of these objects which requires them to be almost always disk resident. With the predicted size and bandwidth requirements of future multi-media objects, this limitation must be resolved if hypermedia systems are to be the wave of future.

This paper describes a placement strategy for the objects of a parallel hypermedia system. The objectives of this strategy are to support a "real-time" display of media objects and to maximize the throughput of the system by uniformly distributing its workload across the processors. In addition, we describe a prefetching mechanism to reduce the response time of the system. Finally, we evaluate our object placement algorithm and its distribution of the workload across the processors.

1 Introduction

A hypermedia system represents and manages information via a network of multi-media objects. It combines different types of information (e.g., text, audio, video) to construct a user-friendly interface for information retrieval. Furthermore, it can organize and manipulate irregularly structured information. Consequently, these systems have become an essential component of many application domains (e.g., education, library information systems, legal research, software engineering, etc.,) and are predicted to be the wave of future.

In a hypermedia application, certain words or phrases (audible or visible) of an object are identified and *hyperlinked* to other objects which describe them in greater detail. Each hyperlink has a frequency of access, defining how frequently it is used to retrieve an object[1]. These frequencies may be updated as a user's pattern of access evolves over time. The collection of objects and hyperlinks form a *hyperlink graph*. This graph can be viewed as a map that shows how information in the system is organized [CONK87] and accessed. In addition, it defines a user's location in the graph and displays the possible objects accessible from that location (i.e., provides a *browsing* capability).

As an example of an application, consider *Compton's Multimedia Encyclopedia* from Britannica Software. According to the publisher [COMP91], it includes the full text of the 19-volume, 5200 article, 8,784,000-word, 1989 edition of the Compton's Encyclopedia; 15,800 pictures, maps, diagrams; 60 minutes of recorded voice and sound; 45 animated sequences; Webster's Intermediate Dictionary; and Josten's word processing program. This is a large volume of information and there are many ways to query it. A typical query would involve the traversal of a hierarchical topic tree which establishes a connection between the relevant pieces of information. So, for example, one can traverse the historical time line (which is one of several topic trees) to hear John F. Kennedy's "Ask not what your country can do for you" speech.

As suggested by this example, a hypermedia system is almost always a read only database. It can be compared to a library managing multiple books and users, where each user accesses the system to retrieve information. While updates are rare, multimedia objects must be displayed in "real-time". By "real-time", we mean a continuous retrieval of an object at the band-

*This research was supported in part by a grant from the USC Faculty Research and Innovation Fund.

[1] One can use Markov chains to compute the frequency of access to an object from the frequency of access associated with its hyperlinks (see Appendix A for details).

width required by its media type. This is a challenging task because certain media types, in particular video, require very high bandwidths. For example, the typical bandwidth required to display a full-screen, full-motion video (without companion audio) is 60 Mbits per second (assuming a 32-bit depth for each pixel). In addition, these objects are usually very large and almost always disk resident. Due to the low bandwidth of the current disk technology (typically rated at 10 Mbits per second), the stand-alone implementation of these systems (based on a workstation) suffers from frequent delays and disruptions, termed *hiccups* [YU89], while the system is displaying an object.

Currently, there are two standard techniques for minimizing the number of hiccups. The first organizes the objects across the disk in order to enhance its bandwidth when retrieving objects [CHRI88]. The second technique reduces the size of a media object in order to decrease the continuous I/O bandwidth required for its retrieval. This can be achieved in two ways: 1) sacrifice the quality of audio and video objects (e.g., use either a low resolution or scanning rate)[2], and 2) apply data compression (e.g., [GALL91, WALL91, HARN91, LIPP89, SIJS91, TINK89]). However, not all of these techniques can satisfy the bandwidth required by most current media objects (let alone the media objects of the future).

In this paper, we propose the use of parallelism to retrieve and display objects of a hypermedia system at their required bandwidth (i.e., real-time display). A parallel hypermedia system is cost effective when multiple users share an application (e.g., multiple users sharing Compton's Encyclopedia). Assuming such a system, this paper describes a placement strategy which declusters and assigns the objects of an application across multiple disks to: 1) use the aggregate bandwidth of several disks to match the bandwidth required for a "real-time" display of an object, and 2) uniformly distribute the workload of an application across the processors in order to maximize the throughput of the system. In addition, we describe a prefetching mechanism to minimize the interval of time elapsed between a user's requests for an object and the time that the system begins to display that object (termed **response time**). Moreover, we present a placement strategy that can distribute the overhead of prefetching uniformly across the processors of the system.

In order to simplify the discussion, we assume a shared-nothing architecture [STON86], however, the algorithms described here can be extended to other architectures (e.g., shared-disk or shared-memory). Briefly, a shared-nothing architecture consists of a number of processors interconnected by a high speed communication network such as a hypercube or a ring. Processors do not share disk drives or random access memory and can only communicate with one another by sending messages using an interconnection network. Furthermore, we assume that the stations used to display objects are independent of the backend processors containing the objects of a hypermedia system as shown in Figure 1 (almost identical to a banking system which consists of a backend database engine and the ATMs attached to it).

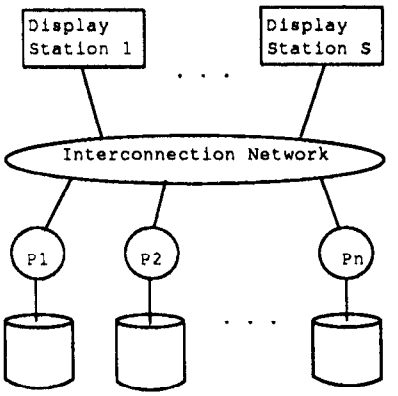

Figure 1: Shared-Nothing Multiprocessor Architecture

The rest of this paper is organized as follows. In Section 2, we describe how the degree of declustering for an object is computed and introduce an algorithm to uniformly distribute the workload of an application across the processors. Section 3 presents a prefetching mechanism to reduce system's response time when an object is requested. Moreover, this section outlines heuristics for distributing the overhead of prefetching uniformly across the processors. In Section 4, we evaluate the accuracy of these heuristics. Our conclusions and future research directions are contained in Section 5.

2 Object Declustering

We decluster [RIES78,LIVN87] each object of an application across several processors in order to use the aggregate I/O bandwidth of these processors to display each object in real-time. In Section 2.1 we describe how to compute the degree of declustering for an object. Subsequently, Section 2.2 describes a placement algorithm for assigning objects to processors in order to uniformly distribute the workload of an application across the processors.

[2] For example, Britannica uses 8-bit, VGA color photos that are soft and blurry. The animated sequences are fuzzy and calling up an animation takes about six seconds [PRES90].

2.1 Degree of Declustering

Assuming that the bandwidth (B) required to display an object x that belongs to media type t is B_t, and the bandwidth of each disk drive is B_{Disk}, we decluster object x across M processors in order to support its bandwidth requirements, where M is defined as:

$$M = \lceil \frac{B_t}{B_{Disk}} \rceil \quad (1)$$

Note that the degree of declustering (M) is a function of the bandwidth required by the media type of an object (i.e., objects of the same media type have identical degrees of declustering). As long as the number of processors in a system is greater than the degree of declustering for the media type with the highest bandwidth requirements, the system can display all objects of an application at the required bandwidth in a single user environment[3].

At first glance, one might attempt to decluster an object across all P processors to: 1) uniformly distribute the workload, and 2) exceed the consumption rate of a user significantly. This strategy, however, suffers from three limitations. First, in a system composed of hundreds to thousands of processors, the time required to activate all the processors might become significant and constitute a significant fraction of the object retrieval time[4] [DEWI88, PATT88, DEWI90, GHAN90]. Second, since each processor must at least read a disk page to retrieve an object, declustering that object across a large number of processors might cause the size of a fragment to be smaller than the size of a disk page at a processor (internal fragmentation), wasting the I/O bandwidth of those processors reading partially empty pages to retrieve that object. Third, when a large number of processors simultaneously send data to a single display station, they might exceed its consumption rate and overflow its memory buffers.

Once the degree of declustering (M) for an object x is evaluated, we form its fragments using round-robin partitioning (see Figure 2). Round-robin partitioning allows a display station to construct and display a portion of x in parallel with the M processors retrieving its remaining portion (i.e., pipelining). If, to the contrary, each fragment was formed by dividing x into M contiguous pieces, the benefits of declustering would be diminished because a display station would have to process and display a fragment of x in its entirety before processing the next fragment.

[3] We have implicitly assumed that the bandwidth of the disk drive is the limiting factor for the system (i.e., bandwidth of the network and network device driver is higher than that of the disk drive).

[4] Equation 1 does not incorporate the overhead of activating multiple processors because this is a one time cost that has no impact on the number of hiccups in the system (i.e., the goal of this paper is to eliminate hiccups).

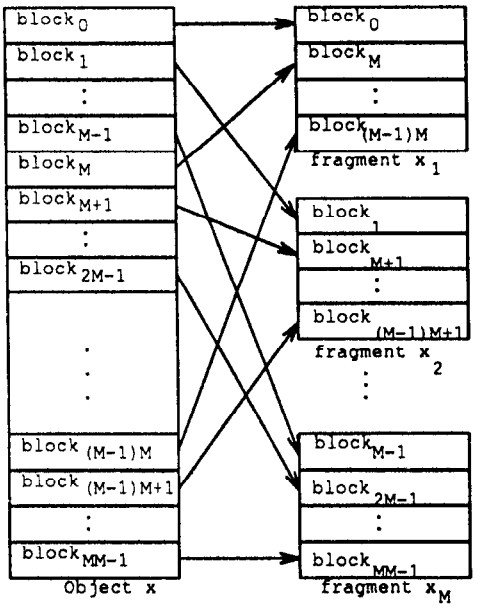

Figure 2: Round-Robin Partitioning of Object x

At a processor, a fragment of an object is stored on contiguous blocks in order to minimize disk seeks. This is primarily because we assume that a user retrieves an object in its entirety.

2.2 Placement Strategy

Once the fragments of each object are formed, they must be assigned to the processors. Our main objective is to distribute the workload of an application uniformly across the processors in order to maximize the throughput of the system. Below, we describe a method that achieves this objective.

Consider a hyperlink graph **G** = (**V**,**A**) where a media object x is defined by a node in **V** and a hyperlink from object x to object y is represented by an arc (x,y) in **A**. Assuming that the hyperlink graph maintains the frequency of access to each object[5], using the terminology of [COPE88] and represent the frequency of access to an object x as *heat*(x) and its size as *size*(x). Next, we represent the *work* imposed on a processor by object x as:

$$work(x) = heat(x) * size(x) \quad (2)$$

Since object x consists of M fragments (x_1, x_2, ..., x_M), the work imposed by each fragment of x (that belongs

[5] We compute the frequency of access to each object using the frequency of access to each hyperlink accessing it (see Appendix A).

(1) Initialize the workload of all the processors to be 0;
(2) Place the P processors in list **L**;
(3) **for** each object x in **V** **do**
(4) decluster x into M fragments;
(5) remove the first M processors from list **L**;
(6) assign the M fragments of x to these processors;
(7) increment the workload of these M processors by the work of each fragment x_i;
(8) insertion sort these processors back into list **L** based on their workload;
(9) **end do**;

Figure 3: A Greedy Algorithm for a Uniform Distribution of Workload

to media type t with bandwidth B_t) on a processor is defined as:

$$work(x_i) = heat(x) * size(x_i) = heat(x) * \frac{size(x)}{\lceil \frac{B_t}{B_{Disk}} \rceil} \quad (3)$$

The workload of a processor is defined as the total work of the fragments (say N) assigned to it:

$$workload(P_i) = \sum_{j=1}^{N} work(frag_j) \quad (4)$$

Using this terminology, the problem of assigning fragments to processors can formally be stated as follows. Assign the objects of an application to processors such that: 1) the fragments of each object are assigned to different processors (i.e. $\forall\, x \epsilon$ **V**, $location(x_i) \neq location(x_j)$ for distinct i and j), and 2) the workload of each processor is the same (i.e., workload(P_n) = workload(P_m), for distinct n and m). The placement algorithm shown in Figure 3 satisfies the first objective while obtaining near optimal workload distributions on most inputs. In this figure, the assignment of M fragments of an object to the first M processors in list **L** ensures that the fragments are assigned to different processors. After adjusting the workload of these processors, by insertion sorting them back into list **L** at step 8, we approximate a uniform distribution of the workload across the processors.

Assuming that F is an upper bound on the total number of fragments in an application, and since a single insertion or deletion from list **L** which consists of P processors is order $log\ P$, the complexity of this algorithm is O($F\ log\ P$).

3 Object Prefetching

In most hypermedia systems, the interval of time elapsed between a user's request for an object to the time that the system begins to display that object might be unacceptable. In this section, we describe a prefetching mechanism that reduces the response time of the system.

While a user is traversing an application, its hyperlink graph can specify the objects that might be retrieved next. Assuming *outneighbors*(x) defines a set of media objects accessible from object x via a hyperlink (i.e., in a graph **G** = (**V**, **A**), $\forall\, y \epsilon$ **V**, $y \epsilon$ *outneighbors*(x) iff $(x, y) \epsilon$ **A**), when a user begins to display object x, one of the objects in *outneighbors*(x) will almost certainly be retrieved at some point later in time. The system can prefetch these objects at two different levels of hierarchy: 1) prefetch from a processor's disk to its memory (eliminate the disk service time), and 2) prefetch from a processor to the user's display station (eliminate both the disk and network service times). Since our emphasis is on the I/O bottleneck, we focus on level one prefetching. However, we have designed algorithms for level two prefetching that are not described in this paper.

In general, it is not beneficial to prefetch each outneighbor of object x in its entirety for two reasons. First, the user will access only one of these objects. Second, the bandwidth consumed to prefetch these objects might interfere with the system's display of objects to other users, reducing the overall throughput of the system. Instead, our prefetching mechanism is designed to retrieve a small fraction of each object while providing the illusion of prefetching each object entirely. Below, we describe a mechanism for achieving this objective and its impact on our declustering algorithm.

3.1 What Portion of an Object is Prefetched?

Without prefetching, when a user requests a neighbor of object x (say y), its response time will consists of: 1) the time for the display station to send a message requesting y to the processors containing y ($Net_{latency}$), 2) the disk service time of these processors to read object y ($Disk_{latency}$), and 3) the network service time for the first page of object y to arrive at the display station ($Net_{latency}$). In current multiprocessor architectures, the disk service time constitutes a significant portion of the response time. In this section, we describe what fraction of y should be prefetched to completely eliminate this factor.

In a single user environment, the $Disk_{latency}$ consists of seek, rotational latency, and transfer times [6].

[6] In this paper, we assume a zero system load and ignore the

In order to eliminate $Disk_{latency}$ time, the system must prefetch a portion of y such that it can overlap the display of this portion with the retrieval of the remainder of y in order to maintain a steady stream of data to a display station. In terms of time, the system should prefetch a portion corresponding to the maximum $Disk_{latency}$ time (termed $Disk_{maxlatency}$). Assuming that the bandwidth required for a real-time display of objects belonging to media type t is B_t, the Volume of Data that should be Prefetched (VDP) is:

$$VDP = Disk_{maxlatency} * B_t \quad (5)$$

For example, if $Disk_{maxlatency} = 25\ msec$ and y is a video object (requires a 60 Mbits/sec bandwidth) whose size is 8 MBytes, only 192 KBytes of y must be prefetched to eliminate its disk latency time (i.e., by prefetching only 192 KBytes (or 2%) of y, we can provide the same response time as when y is prefetched in its entirety).

From this point on, we term the prefetched portion of an object its *head* and the rest of it as its *tail*.

3.2 Degree of Declustering for the Head

A major assumption thus far has been that the bandwidth of a disk drive is the limiting factor for a real-time display of an object. However, the bandwidth of the network device driver of a processor is not that much higher[7]. Thus, while the head of an object might have already materialized at the memory of a processor, the bandwidth of its network device driver might not be high enough to support a real-time display of that portion. In order to resolve this limitation, we decluster the head of an object across several processors. Assuming that the required bandwidth for a real-time display of an object x of type t is B_t, and the bandwidth of a network device driver is $B_{Net-Interface}$, we decluster the head of object x into H fragments, where H is defined as:

$$H = \lceil \frac{B_t}{B_{Net-Interface}} \rceil \quad (6)$$

Note once again, that the degree of declustering for the head is a function of the media type. The degree of declustering for the tail of an object is still determined using Equation 1. Furthermore, since we assumed that the bandwidth of the network interface is significantly greater that that of the disk ($B_{Net-Interface} \gg B_{Disk}$), the degree of declustering for the head of an object is lower than that for its tail (i.e., $H < M$).

3.3 When Prefetching Should be Avoided

At this point, one might be lead to believe that prefetching is beneficial under all circumstances. However, this is not true for an object that has a lower display time than the prefetch time of its outneighbors. In this case, prefetching must be avoided as it will only degrade the performance of the system. For example, if the display time of object x is 0.05 seconds and the time to prefetch each object in outneighbors(x) is at least 5 seconds, then a user will display x before the system can materialize the prefetch portion of any object in outneighbors(x). Furthermore, once the user elects to retrieve an object y in outneighbors(x), there will be many hiccups because the head fragments of y are disk resident while these fragments were formed based on the assumption that they are memory resident (using Equation 6). Moreover, in this case, prefetching wastes a lot of resources because the system might continue to prefetch objects in outneighbors(x) even though the user has already committed to retrieve only one of these objects. In this case, prefetching is an overhead with no benefits whatsoever. As a solution, we do not prefetch objects in outneighbors(x) whose prefetch-time is significantly greater than the display time of x. This is achieved by analyzing the hyperlink graph and marking the objects with a lower display time than the prefetch time of its outneighbors.

3.4 Summary

In summary, the objects of a hypermedia application are placed across the processors as follows. First, the objects of an application are grouped according to their media type and bandwidth requirements. For each object in a group, we evaluate its prefetch portion using Equation 5 and distinguish its head from its tail. Using Equation 1 and 6, we evaluate the degree of declustering for the head and tail of each object (i.e., determine M and H). The tail fragments are then assigned to processors using the algorithm outlined in Figure 3. Below, we describe how the head fragments are assigned to processors. **From this point on, whenever we refer to fragments of an object we imply the fragments that constitute the head of that object** (since its tail fragments have already been assigned).

time a request might spend in the disk queue of a processor waiting for service. The extensions of this work to incorporate a system load is a part of our future research directions.

[7] While the aggregate network bandwidth of a multiprocessor is very high due to the recent advent of hypercube and mesh interconnections, the bandwidth of a network device driver is usually rated at 2.8 MBytes/Second.

3.5 Placement of the Prefetched Fragments

Ideally, we would like to assign the fragments of objects in outneighbors(x) such that the overhead of prefetching is evenly distributed across the processors. Furthermore, prefetching should not interfere or compete for resources with the current display of an object x. The problem can formally be stated as follows. Assign the fragments of objects in outneighbors(x) such that: 1) the fragments are evenly distributed across the processors, and 2) the fragments are assigned to processors different than those containing the tail of x.

We have designed two placement algorithms for assigning head fragments to processors.[8] The first assigns objects on an individual basis while the second assigns objects on a set basis.

Before describing our algorithms, we define the term *sibling* that is used repeatedly in this section. Given an object x, set Z consists of all objects with an arc to x. For example, given o_4 in Figure 4, set Z consists of o_7 and o_3. For each object y in Z, if there is an arc from y to w, where w is not x, then w is a sibling of x (i.e., $siblings(x) = \{w : (y,x) \in \mathbf{A}, (y,w) \in \mathbf{A}, \text{and } w \neq x\}$. Consequently, the siblings of o_4 are: o_1, o_2, o_5, o_6, o_8, o_9, and o_{10}. The siblings of o_1 are: o_2, o_4, o_5, and o_6. Intuitively, the siblings of x are all the possible objects that might be prefetched along with x.

3.5.1 Object-Based Assignment

With Object-Based (OB) assignment, the fragments of each object are analyzed and assigned on an individual basis. In this context, we must address two issues. First, in what order should the system assign the objects of an application. Second, given an object, which processors should contain its fragments. Below, we describe our heuristics for resolving these issues.

Our heuristic for choosing an object analyzes a graph and associates a weight to each object. The objects are assigned based on a Heaviest Object First (**HOF**) heuristic. The weight of an object is the total size of the fragments that constitute its *siblings*. For example, if the degree of declustering for the head of each object in Figure 4 is three, then the weight of o_4 is 21 while that of o_1 is 12.

The rationale behind HOF is as follows. The head fragments of a heavy object (say x) is prefetched along with many other objects in the system. Consequently, x has a high number of constraints associated with it because these fragments should be uniformly distributed across the processors in order to distribute the overhead of prefetching uniformly. By assigning x first, one can choose from a large number of available processors and propagate its constraints to direct the assignment of objects with fewer constraints (HOF is based on the *Constraint Propagation* [SUSS80, WALT75] heuristic used for problem solving in AI). Thus, for example, HOF assigns o_4, o_5, and o_6 of Figure 4 first, before assigning any other object.

[8] We speculate that obtaining an optimal solution to this problem is NP-hard. However, we have not yet been able to reduce this problem to one of the well known NP-hard problems.

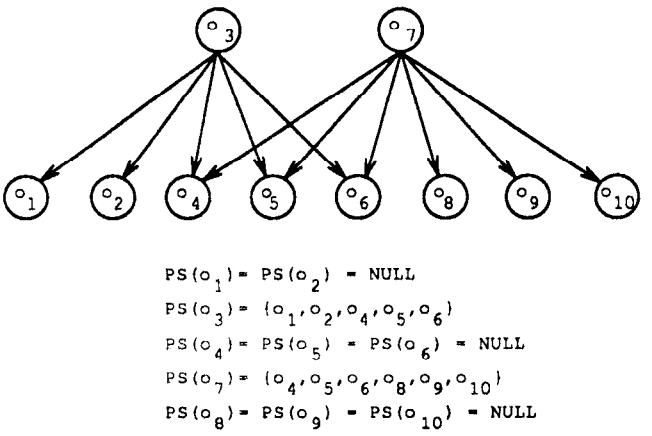

Figure 4: An Example Hyperlink Graph

In Section 4, we quantify the advantages of using HOF as compared to randomly choosing objects. In addition, we analyze a Lightest Object First (**LOF**) heuristic and characterize the tradeoffs associated with assigning heavy objects last.

Given an object x, its head fragments are assigned to the processors suggested by its siblings. Each of its siblings suggests a set of processors different than those containing that sibling. If a sibling is not assigned to a processor, it suggests all the processors in the system. We assign x to the Most Frequently Suggested Processor (**MFSP**). This heuristic approximates a uniform distribution of the overhead of prefetching across the processors.

It is important to note that MFSP is the mechanism that propagates the constraints of HOF to direct the assignment of other objects. To illustrate, if o_4 is the first object to be assigned, its siblings suggest each processor in the system as frequently allowing o_4 to be assigned to any processor. However, after o_4 is assigned, the set of processors that can contain o_1 is limited because o_1 is a sibling of o_4 and o_4 will not vote for any processor containing itself.

3.5.2 Set-Based Assignment

The set-based assignment analyzes the graph and forms the **prefetch-set** of each object x. The prefetch-set of x (denoted as either prefetch-set(x) or PS(x)) consists

of a set of objects accessible from x via a hyperlink. It defines the objects that are prefetched every time a user displays object x. The set-based algorithm analyzes and assigns the fragments of each prefetch-set one at a time. This is a greedy strategy because it focuses on a single prefetch-set and tries to distribute the overhead of prefetching as evenly as possible for that set.

Since each object has a prefetch-set, a hyperlink graph consisting of |V| objects has |V| prefetch-sets. Figure 4 shows an example hyperlink graph and its corresponding prefetch-sets. Note that an object might appear in several prefetch-sets. Consequently, once an object in a set is assigned, it enforces constraints on the processors that may contain the objects of other sets that include it. For example, objects o_4, o_5, and o_6 of Figure 4 are common to both PS(o_3) and PS(o_7). Assuming that PS(o_7) is assigned before PS(o_3), when assigning objects in PS(o_3), the assignment of o_4, o_5, and o_6 imposes constraints on the processors that may contain o_1 and o_2. This is because we insist on a uniform distribution of the fragments of PS(o_3) across the processors while some of its object (i.e., o_4, o_5, and o_6) have already been assigned by PS(o_7).

The order in which prefetch-sets are assigned affects the distribution of the overhead of prefetching. For example, if PS(o_7) is assigned before PS(o_3), when assigning PS(o_3), two objects (o_1 and o_2) can be used to compensate for any uneven distribution that might have resulted from the assignment of objects o_4, o_5, and o_6. On the other hand, if PS(o_3) is assigned before PS(o_7), more objects (o_8, o_9, and o_{10}) can be used to compensate for any uneven distribution. In reality, there is a higher degree of freedom associated with a set that consists of a large number of objects with indegree one. This is because these objects can be used to compensate for any uneven distribution in a set. Thus, we would like to assign sets with a Low Degree of Freedom First (i.e., a LDFF heuristic), in order to satisfy their constraints and propagate them to direct the assignment of sets with fewer constraints[9].

The degree of freedom associated with a prefetch-set is computed as follows. First, we determine the **indegree signature** of each set. An indegree signature is a vector consisting of |V| elements $(i_1, i_2, ..., i_j, ..., i_{|V|})$. The jth element of this vector (i_j) defines the number of objects with j incoming arcs (i.e., indegree). Thus, i_j specifies that there are i objects that appear in j prefetch-sets. For example, the indegree signature of PS(o_3) is (2,3). The first number specifies that there are two objects (o_1 and o_2) that appear in only one prefetch-set (i.e., $PS(o_3)$) while the second specifies that there are 3 objects (o_4, o_5, o_6) that appear in two prefetch-sets (i.e., $PS(o_3)$ and $PS(o_4)$). Given an indegree signature of a prefetch-set, we define its Estimated Degree of Freedom (EDF) as:

$$EDF = \sum_{j=1}^{|V|} i_j \times \frac{1}{j} \quad (7)$$

For example, the expected degree of freedom associated with prefetch-set(o_3) is:

$$(2 \times \frac{1}{1}) + (3 \times \frac{1}{2}) = 2.5$$

The order in which prefetch-sets are assigned is determined by the LDFF heuristic. Given a prefetch-set, the objects in that set are assigned based on the HOF heuristic. The fragments of an object are assigned to the processors that retrieve the Lowest Volume of data for the Prefetch-sets already assigned (**LVP** heuristic).

4 Evaluation of the Object Placement Algorithm

In this section, we evaluate the alternative heuristics for assigning fragments to processors. We begin by describing a criteria for conducting this evaluation. Next, we analyze the heuristics based on a syntactically generated hyperlink graph. Finally, in Section 4.3, we evaluate the heuristics using a hyperlink graph of an actual hypermedia application.

4.1 Evaluation Criteria

A major objective of our placement strategy is to uniformly distribute the overhead of prefetching across the processors. This overhead is defined as the volume of data retrieved on behalf of a prefetch-set. Thus, we use the variance in the volume of data retrieved by each processor for a prefetch set (v_{PS}) as the metric for evaluating our alternative assignment strategies. For example, assume that each object o_i in Figure 4 is declustered into three fragments ($o_{i,1}, o_{i,2}, o_{i,3}$). Furthermore, assume that the head fragments of these objects are assigned to a four processor system as shown in Figure 5. The objects in prefetch-set(o_3) are o_1, o_2, o_4, o_5, o_6. For this prefetch-set, the assignment of Figure 5 results in the retrieval of five fragments by processor 1, four fragments by processor 2, five fragments by processor 3, and one fragment by processor 4. Assuming equi-sized fragments, the variance in the volume of data prefetched by each processor is skewed and $v_{PS(o_3)}$ is 2.7. This variance is evaluated using a standard statistical method that is described in Appendix B.

[9] Section 4 compares the performance of LDFF with a Highest Degree of Freedom First (**HDFF**) heuristic.

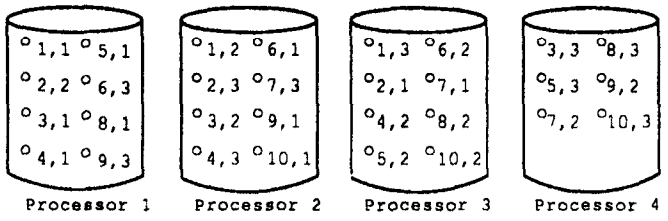

Figure 5: Sample Placement of Head Fragments over a Four Processor System

After evaluating the variance of each prefetch-set, we compute the Average Variance for a graph according to its frequency of access (i.e., heat):

$$Avg.\ Variance = \sum_{x \epsilon V} heat(PS(x)) \times v_{PS(x)} \quad (8)$$

The heat of $PS(x)$ (or its frequency of access) is equivalent to the heat of object x because every time x is accessed, the objects in $PS(x)$ are also retrieved. In the following experiments, we assume a uniform distribution of access to the objects of an application (i.e., $heat(PS(x)) = heat(x) = \frac{1}{|V|}$, where |V| is the total number of objects in the application), and Equation 8 was used to compute the average variance in the overhead of prefetching.

4.2 Evaluation of Different Heuristics

In these experiments, we evaluated the alternative heuristic combinations to determine how closely each estimates an optimal solution. Finding an optimal assignment for a graph requires an exhaustive search of the possible solution space which is extremely large and not computable in a reasonable amount of time. Consequently, we designed a syntactic graph with a known optimal solution (average variance of zero) for a sixteen processor system. This graph consisted of fifty nodes, with 50% of its node having an indegree one. Furthermore, we declustered the head and tail of each object into 3 and 6 fragments respectively (i.e., $H = 3$ and $M = 6$). Below, we analyze the accuracy of heuristics for each of set-based and object-based assignments.

4.2.1 Accuracy of Alternative Set-Based Heuristics

Table 1 presents the accuracy of different set-based heuristics for a syntactically generated hyperlink graph. The first three columns of this table contain the different heuristic combinations, while the fourth contains the average variance in the volume of data prefetched by a sixteen processor system for a given heuristic combination. The term "random" in either the prefetch-set or object column specifies a random assignment of a prefetch-set or an object. Similarly, "random" in the processor column denotes a random assignment of the fragments of an object to processors[10]. The term "round-robin" in the processor column refers to a round-robin assignment of the fragments of an object to processors.

The first six rows of Table 1 demonstrate that if the fragments of a prefetch set are not assigned intelligently, regardless of how accurately one choose the prefetch-sets or objects of a prefetch-set, the distribution of the overhead of prefetching will be skewed. The heuristic for choosing processors (i.e., LVP) is quite effective by itself (see row 7). However, note that LDFF in combination with HOF and LVP can obtain an optimal solution while LVP by itself cannot converge to such a solution.

Rows 10, 11, and 12 show no difference between HDFF and LDFF heuristics because LVP eclipses the difference between these two heuristics. However, observe from rows 7, 8, and 9 that LDFF results in a lower average variance as compared to either HDFF or a random assignment of prefetch-sets.

The last column of Table 1 establishes the time complexity for each heuristic. Table 3 contains the definition of terms used in that column. With the LDFF and HDFF heuristics, forming the neighbor-sets and computing each object's weight requires $|A|$ traversals. Since sorting the prefetch-sets of an application is order $N\ log\ N$, the algorithms that use either LDFF or HDFF have at least a complexity of $O(|A| + N\ log\ N + NH)$, where NH is the total number of fragments.

4.2.2 Accuracy of Alternative Object-Based Heuristics

Table 3 contains the accuracy of alternative object-based heuristics. The results demonstrate that the combination of HOF and MFSP heuristics approximate the optimal solution significantly closer than the other alternatives. The MFSP heuristic is very effective by itself (compare row 7 with rows 1-6). However, HOF is instrumental as it reduces the average variance by a factor of four as compared to MFSP all by itself. In addition, note that LOF heuristic degrades the accuracy of MFSP strategy (compare row 7 with 9). A major conclusion to be drawn here is that assigning an object with highest number of constraints first is beneficial.

[10] The experiments with a random assignment were repeated many times in order to establish a 95% confidence interval on the obtained results.

	Heuristics Used			Avg. Variance in Overhead of Prefetching	Time Complexity
	PS	Object	Processor		
1	Random	Random	Random	443.8423	$O(NH)$
2	LDFF	Random	Random	443.2335	$O(\|A\| + NlogN + NH)$
3	LDFF	HOF	Random	440.0519	$O(\|A\| + NlogN + NH)$
4	Random	Random	Round-Robin	113.5160	$O(NH)$
5	LDFF	Random	Round-Robin	117.6105	$O(\|A\| + NlogN + NH)$
6	LDFF	HOF	Round-Robin	60.8869	$O(\|A\| + NlogN + NH)$
7	Random	Random	LVP	1.0655	$O(N(H+P)logP)$
8	LDFF	Random	LVP	0.8372	$O(\|A\| + NlogN + N(H+P)logP)$
9	HDFF	Random	LVP	1.6896	$O(\|A\| + NlogN + N(H+P)logP)$
10	**LDFF**	**HOF**	**LVP**	**0.0000**	$O(\|A\| + NlogN + N(H+P)logP)$
11	HDFF	HOF	LVP	0.0000	$O(\|A\| + NlogN + N(H+P)logP)$
12	LDFF	LOF	LVP	0.0000	$O(\|A\| + NlogN + N(H+P)logP)$

Table 1: Set-Based Assignment

	Heuristics Used		Avg. Variance in Overhead of Prefetching	Time Complexity
	Object	Processor		
1	Random	Random	446.6966	$O(NH)$
2	HOF	Random	438.6979	$O(\|A\| + NlogN)$
3	LOF	Random	440.5927	$O(\|A\| + NlogN)$
4	Random	Round-Robin	494.6905	$O(NH)$
5	HOF	Round-Robin	60.8869	$O(\|A\|HlogP + NlogN)$
6	LOF	Round-Robin	114.1636	$O(\|A\|HlogP + NlogN)$
7	Random	MFSP	10.4345	$O(\|A\|HlogP)$
8	**HOF**	**MFSP**	**0.0000**	$O(\|A\|HlogP + NlogN)$
9	LOF	MFSP	68.497738	$O(\|A\|HlogP + NlogN)$

Table 2: Object-Based Assignment

Term	Definition
N	Number of objects in an application
$\|A\|$	Number of arcs in a graph
P	Number of processors
H	Degree of declustering for the prefetch portion

Table 3: List of terms used for establishing time complexity and their respective definitions

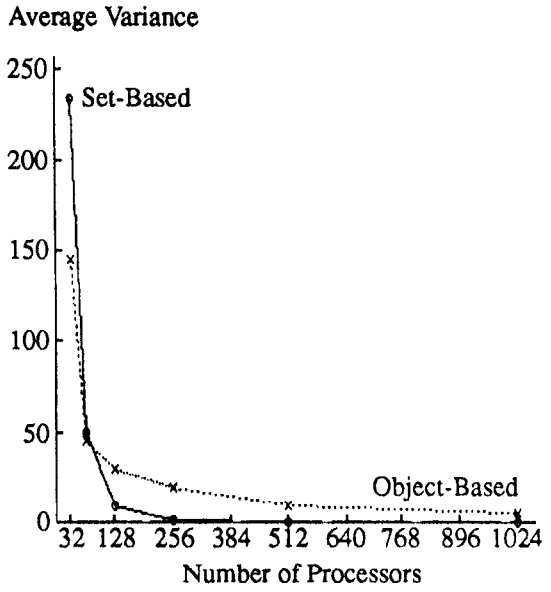

Figure 6: Accuracy of Neighbor-Based versus Object-Based Algorithms

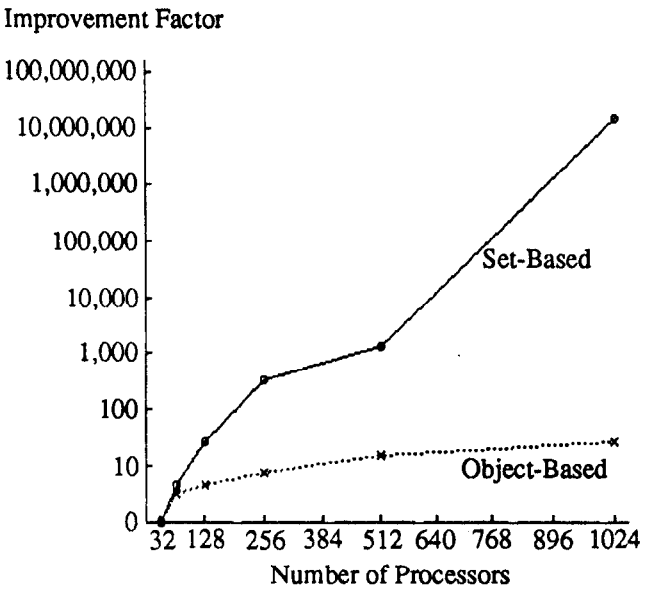

Figure 7: Factor of Improvement as Compared to a 32 Processor System

4.3 A Comparison of Object-Based with Set-Based Algorithms

Finally, we compared the best object-based heuristic combinations (HOF, MFSP) with that of set-based assignment (LDFF, HOF, LVP). Each heuristic assigned a hypermedia application termed "John Cocke: A Retrospective By Friends". This application was authored at the Multimedia laboratory of the IBM TJ Watson Research Center and consisted of audio, video and bitmap media objects. For the purposes of our experiment, we restricted this application to video objects only (237 objects). The degree of declustering for the tail of each object was set to six while that of its head was set to three (i.e., $M = 6$ and $H = 3$).

In our experiments, we varied the number of processors in a system from 32 to 1024 while observing the average variance of both set-based and object based assignments. Figure 6 contains the results of these experiments. As we increased the number of processors in the system, the average variance of both strategies approached zero because the larger number of processors increases the degree of freedom associated with the assignment procedure. To justify this claim, in a different experiment, we increased the degree of declustering of each object proportional to the number of processors in the system and observed no drop in the average variance of either algorithm.

When the number of processors is less than sixty-four, object-based assignment has a lower variance than set-based assignment. However, with more processors, set-based assignment starts to outperform object-based assignment. Fewer processors translates into a larger number of constraints on the assignment procedure. In this case, object-based assignment can propagate constraints more effectively since it propagates them at a lower granularity level, namely that of an object, where constraints can be satisfied and propagated. While set-based assignment propagates constraints at the granularity of a set (significantly larger than an object), where constraints are neither satisfied nor propagated.

However, set-based assignment obtains a very low variance when the number of processors is large enough to enable constraints to be propagated among sets. In order to demonstrate this affect, Figure 7 shows the improvement factor of each assignment strategy relative to a 32 processor system as a function of the number of processors in the system. The y-axis in this figure is logarithmic. This figure shows that as the number of processors is increased, set-based assignment can propagate the constraints and estimate an optimal solution significantly closer than the object-based assignment.

5 Conclusion and Future Research Directions

In this paper, we described a placement strategy for the objects of a parallel hypermedia system. This strategy declusters each object across multiple processors in order to display it in real-time. Furthermore, it maximizes the throughput of the system by distributing the workload of a hypermedia application uniformly across the processors. We also described a prefetching mechanism to reduce the response time of the system. In addition, we proposed and evaluated the accuracy of several heuristics for distributing the overhead of prefetching uniformly across the processors. Our evaluation indicates that the proposed placement heuristics distribute the overhead of prefetching more uniformly than both round-robin and random object placement strategies.

This work is pioneering and we are currently extending it in several directions. First, we are augmenting our placement algorithm to support a skewed distribution of access to the hyperlinks of an application. Second, we are extending our declustering strategy and prefetching mechanism to incorporate different system loads (an assumption of this paper is zero system load). A system load can result in certain amount of wait-time for service which can have a significant impact on the real-time display of an object.

Third, we are designing several algorithms for dynamic on-line re-organization of objects to maintain an even distribution of the workload in the presence of a user's changing pattern of access to the objects of a hypermedia application.

Finally, we are implementing a simulation model of a parallel hypermedia system to quantify the performance of the declustering strategy and prefetching mechanism proposed in this paper.

Appendix A Frequency of Access to Objects

We compute the frequency of access to the objects of a hypermedia application based on the frequency of access to its hyperlinks. In this appendix, we provide a stochastic method for performing this computation.

Given a hyperlink graph $\mathbf{G} = (\mathbf{V},\mathbf{A})$, we construct its Markov chain $\mathbf{M} = (\mathbf{S},\mathbf{T})$, where each object in $\mathbf{V}$ is defined as a state in $\mathbf{S}$ and each hyperlink in $\mathbf{A}$ is represented by a transition from one state to another. We define an $n \times n$ probability transition matrix,

$$\mathbf{P} = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,n} \end{bmatrix}$$

where n is the number of states in the Markov chain. The entry $p_{i,j}$ specifies the probability of using a hyperlink from object i to j provided that the user is currently viewing object i. For example, assuming a uniform probability of access to the outneighbors of object i, the transition probability from object i to j is:

$$p_{i,j} = \frac{1}{outdegree(j)}$$

where $outdegree(j)$ is the number of outgoing hyperlinks from object j. Note that in this case, the probabilities for each row must sum up to 1.

Assuming that the term x_i of vector $\mathbf{x} = [x_1, x_2, \ldots, x_i, \ldots, x_n]$ specifies the frequency of access to each object i of a hypermedia application, then x_i is computed by finding a solution to $\mathbf{x}$ from the following system of equations:

$$(\mathbf{P} + \mathbf{1}_{n\times n} - \mathbf{I})\mathbf{x} = \mathbf{1}_{n\times 1}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{1}$ is a matrix whose elements are all 1.

Appendix B Variance in Volume of Data

In this section, we describe how the variance in volume of data prefetched by each processors is computed.

Assuming a system composed of P processors, and a hyperlink graph whose fragments have already been assigned, we compute the volume of data prefetched by processor i ($1 \leq i \leq P$) on behalf of a prefetch-set(x) (termed $VOL_{PS(x),i}$). Next, we compute the average volume of data that should ideally be prefetched by each processor had the fragments of PS(x) been distributed uniformly across the processors (termed $\overline{VOL_{PS(x)}}$)[11]. Using this information, we compute the variance in the volume of data prefetched by each processor for a given prefetch set ($v_{PS(x)}$), as follows:

$$v_{PS(x)} = \sum_{i=1}^{P} \frac{(VOL_{PS(x),i} - \overline{VOL_{PS(x)}})^2}{P} \qquad (9)$$

[11] When the number of fragments cannot be divided equally among the processors, we adjust the variance to compensate for this case.

References

[CHRI88] S. Christodoulakis. Performance Analysis and Fundamental Performance Trade Offs for CLV Optical Disks. Tech. Report CS-88-06, University of Waterloo, 1988.

[COMP91] Brittanica Inc. Compton's Multimedia Encyclopedia. Encyclopedia Brittanica, Inc. 1991.

[CONK87] E. Conklin. Hypertext: An Introduction and Survey. IEEE Computer, September 1987.

[COPE88] G. Copeland, W. Alexandar, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proceedings of the 1988 ACM SIGMOD Int'l Conf. on Management of Data*, May 1988.

[DEWI88] D. DeWitt, S. Ghandeharizadeh, D. Schneider. A Performance Analysis of the Gamma Database Machine. In *Proceedings of the 1988 ACM SIGMOD Int'l Conf. on Management of Data*, May 1988.

[DEWI90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[GALL91] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. In *Communications of the ACM*, April 1991.

[GHAN90] S. Ghandeharizadeh and D. DeWitt. A multiuser performance analysis of alternative declustering strategies. In *Proceedings of the 6th Data Engineering Conference*, February 1990.

[HARN91] K. Harney, M. Keith, G. Lavelle, L. Ryan, and D. Stark. The i750 Video Processor: A Total Multimedia Solution. In *Communications of the ACM*, April 1991.

[LIPP89] A. Lippman, and W. Butera. Coding Image Sequences for Interactive Retrieval. In *Communications of the ACM*, July 1989.

[LIVN87] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. In *Proceedings of the 1987 ACM SIGMETRICS Int'l Conf. on Measurement and Modelling of Computer Systems*, May 1987.

[PATT88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Int'l Conf. on Management of Data*, May 1988.

[PRES90] L. Press. Compuvision or Teleputer? *Communications of the ACM*, September 1990.

[RIES78] D. Ries and R. Epstein. Evaluation of distribution criteria for distributed database systems. UCB/ERL Technical Report M78/22, UC Berkeley, May 1987.

[SIJS91] F. Sijstermans and J. van der Meer. CD-I Full-Motion Video Encoding on a Parallel Computer. In *Communications of the ACM*, April 1991.

[STON86] M. Stonebraker. The Case for Shared-Nothing. In *Proceedings of the 1986 Data Engineering Conference*, Los Angeles, CA, 1986.

[SUSS80] G. Sussman and G. Steele. CONSTRAINTS: A Language for Expressing Almost-Hierarchical Description. In *Artificial Intelligence*, 1980.

[TINK89] M. Tinker. DVI Parallel Image Compression. In *Communications of the ACM*, July 1989.

[WALL91] G. Wallace. The JPEG Still Picture Compression Standard. In *Communications of the ACM*, April 1991.

[WALT75] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In *The Psychology of Computer Vision*, P. Winston (Ed.,), McGraw Hill, New York, 1975.

[YU89] C. Yu, W. Sun, D. Bitton, Q. Yang, R. Brunno, and J. Tullis. Efficient placement of audio on optical disks for real-time applications. In *Communications of the ACM*, July 1989.