Towards An Efficient Management Of Objects In A Distributed Environment

A. El Habbash  J. Grimson  C. Horn

Department of Computer Science, Trinity College Dublin, Ireland

Abstract

A prototype of an object oriented system implemented in C.Prolog is described in this paper. Its main objective is to demonstrate system features that would support efficient management of objects and object oriented databases in a persistent and distributed environment. Mechanisms at the low level of the system were considered to support object distribution, mobility control, and configuration management in a simple and uniform way. Objects exist in clusters, which are transparent to the applications. The prototype is a framework for a self organizing object oriented distributed system.*

1 Introduction

Locality of reference is a basic feature of most programs. Program behaviour has been characterized by phases or bounded locality intervals in a hierarchical way, which to a large extent reflects the natural construction of these programs [10, 12]. Similar trends have been noticed for files and databases with respect to data access [11, 18, 13, 15, 21, 6]. In object oriented modelling, the concept of dependent objects (objects that cannot exist without a host object, and cannot be shared with other objects outside the host) suggests that these objects must form a unit of locality [7]. Results of other experiments in object oriented programming recommend clustering the object properties together as a unit of locality [2, 8, 20].

In addition to this natural phenomenon of locality which was the main reason behind the success of the virtual memories in general, much research has been carried out to extend this phenomenon and force stronger locality. Besides the high level techniques of modelling, and program structuring, there have been low level techniques to reorganize the program code [4, 9], or to cluster the database records [1, 14, 17, 23, 16]. Few experiments have been carried out to enhance locality in object oriented environments. Both static references and dynamic reference behaviour have been considered [19, 22], and it was noticed that, in most cases of reorganization, better performance was achieved.

Most of the studies of locality and reorganization considered paged virtual memories. A memory block was considered as the unit of clustering, and the objective was to co-locate within a memory block, the objects that would be referenced together, so as to minimize page faults in memory, and accordingly minimize page I/O traffic between memory and external storage.

Computer networks and distributed systems add more dimensions for clustering; communication overhead, system reliability, load balancing and nodal autonomy must be considered. Objects must be clustered on a nodal basis before being clustered within a memory block. The file assignment problem (FAP) has been a rich area for research; however, the solutions reached were computationally expensive [3].

Persistent programming encouraged object reuse as well as sharing by different programs or different users. This case gave rise to an environment that would lead to different, changing, and possibly conflicting access requirements. This is similar but on a larger scale to the environment encountered in traditional database systems, where a database administrator (DBA) is required to monitor and control the access to the database, and also to adapt the database to the different access requirements. This may not be a feasible solution in the environment under consideration.

The environment considered in this paper is a distributed object oriented system supporting persistent programming and object databases. The basic features of COMANDOS** [5], were considered in developing this prototype. The prototype is used as a framework to emphasize the main features of the system that would simplify the management task regarding the control of locality, application configuration and garbage collection. The objective is to have a system which is self organizing. The top level of management is proposed to be an expert system that would interact with the system through monitoring information and management tools (not discussed in this paper). The implementation language chosen is C.Prolog.

2 The Basic Model

The model represents a distributed, object oriented system built on top of a number of processing nodes connected via a communication network. Some of the nodes are associated with a large external storage (called storage sites). The external storage consists of permanent storage devices like disk packs, disk partitions,

* This work is partially supported by ESPRIT contracts 634 and 2071

** An ESPRIT project for the Construction and Management of Distributed Open systems. The kernel of COMANDOS, an object oriented language OSCAR, and an implementation of the object database are being developed within the Distributed Systems Group (DSG) at Trinity College, Dublin.
objects. These will be referred to as physical containers. At the computational level, the storage system is modeled as a number of logical containers. The logical container is uniquely identified in the system. It consists of one or more physical containers which can be located at one or more storage sites. Persistent objects are stored in the storage system. Objects that can be referenced system wide are called public objects. Each public object is identified by a unique name called its public identifier (PID). The PID consists of the logical container identifier, and a generation number (simply a time stamp or a serial number) within the logical container and some other control information. Apart from the size limitation, the number of the public objects in the logical container is limited by the width of the PID.

Objects in a logical container could be physically replicated at more than one physical container. Objects in the model are typed entities. The type defines the interface to its instances which comprises the public properties and the public operations (methods). Properties can be of variant types and can be multivalued. Types are related in a subtyping relationship and an inheritance hierarchy. An instance of a type is implemented as an instance state comprising the public properties and possibly some other private properties known only to the implementation. The type implementation implements the operations of the type which manipulate the instance state. There could be multiple implementations of the same type. The model also supports database applications. Instances of the same type can be related by an object called a class. The class combines the notion of intension defining the properties of the class instances together with the notion of extension (a collection of objects in the class). The class also defines for its instances the storage schema and the allocation schema.

The computation in the model starts by running a job which provides an environment for the computation. It starts by creating a context within an address space at an initial node; then as objects are referenced they are mapped to the context (retrieved from the storage system or migrated from another context). The job may also diffuse to some other nodes by creating contexts there. Operation invocation is carried out as in the following steps:

a- Locate the object header and the instance data.

b- Locate the object implementation (referenced by the object header in step a above).

c- Select the operation provided and carry out the invocation using the object state and the supplied parameters.

3 Design Issues

3.1 Design Considerations

At the initial stages of the model implementation, there were some problems which influenced the design decisions. The problems which are outlined below, led to the introduction of system units, at the kernel level, transparent to the application developer, called clusters. The cluster, as will be explained later, can be seen as an image of a memory partition containing objects as well as some control and management information. The problems encountered can be summarized as:

a- Object management could be a serious problem due to the fine granularity of objects as well as object mobility. The size and the overhead of keeping access, control and other management information for each object could be overwhelming. The cluster could provide more manageable units.

b- Object invocation as explained above (section-2) was not expected to be efficient due to the large overhead involved in mapping each requested object on demand to the running context. The cluster could be considered as a cache of objects that could be mapped as one unit.

c- Large scale distribution, persistence, and inter objects references in such an environment make global garbage collection very expensive, if not (practically) infeasible. On the other hand, garbage collection within a cluster which is confined to a single node could be feasible.

d- Object identifiers are limited within the logical container; thus they should not be unnecessarily wasted, apart from being expensive to generate and use as a reference. Clusters could offer a simple naming domain to the objects unknown outside the cluster, and that would reduce the need for generating PID's.

3.2 Cluster Design and Implementation

The cluster is basically a group of objects. Each object must exist in some cluster and thus will be identified by a local identifier (LID) which is valid only within the cluster. Objects in a cluster can be referred to by other objects in the cluster (intra cluster references) using the LID as a local reference, and can be also referenced by objects external to the cluster (inter cluster references) using the PID as a global reference. If locality of reference is assumed, most the objects in the cluster will not need to be given PID's since they will be referenced only locally (private objects). Only objects which are referenced globally will be assigned PID's.

Objects in the cluster are grouped into regions. The regions are in general contiguous areas in the cluster. They can be used for page prefetching, if organized according to their access behaviour, or could be used for the local garbage collection within the cluster, if organized as age groups (generation scavenging). Global garbage collection is still a problem. This is handled by providing a storage hierarchy (which will be called archival hierarchy) such that objects which are not used during an observation period, or suspected to be garbage, are moved up in the hierarchy, to be finally archived off-line or possibly destroyed later. Every cluster will be associated with a higher level cluster in the archival hierarchy called the archival cluster.

3.3 Cluster Structure

The cluster contains the following areas, see (fig.1):

- cluster header (CH).

It contains offsets for the different areas in the cluster (as described below) and some other general, control, management, security, and statistical information about the cluster. It also has a reference to the associated archival cluster.
Fig. 1 - Cluster Layout

<table>
<thead>
<tr>
<th>Object pid</th>
<th>Cluster</th>
<th>Location or Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid1</td>
<td>C0</td>
<td></td>
</tr>
<tr>
<td>pid2</td>
<td>C0</td>
<td></td>
</tr>
<tr>
<td>pid12</td>
<td>C1</td>
<td></td>
</tr>
</tbody>
</table>

Object entry

Cluster entry

Fig. 2 - Logical Container
b- Private table(PrT).
Each object in the cluster is associated with an entry in this table which is called the object header. The subscript of the object header in the array is in fact the LID of the object in the cluster. Each entry in the table contains the following fields:

- Offset: indicates the location of the object from some reference point in the cluster.
- Size: The size of the object.
- GN: This entry contains the generation number part of the object PID (valid only for the objects which are globally referenced).
- Imp: This is a reference to the implementation object.
- Mig.flag: This is the migration flag, if set then it means that the object has migrated the cluster.
- SC: Statistical and control information.

c- Export table(ExT).
This is a hash table containing an entry for each public object in the cluster. It is searched by hashing the PID. It is used primarily to translate the global reference using PID into an LID in the cluster. The main fields are:

- Subscript: This is the LID of the public object.
- SC: Statistical and control information.

d- Import table(ImT).
This table contains an entry for each external public object referenced by objects in the cluster. Its primary function is to give a quick hint about the location of the external object as well as some control and statistical information. Each entry contains fields as:

- Ch: Cluster hint refers to the cluster identifier (CID) of the containing cluster (this is considered only as a hint that can be used to access the given cluster if it is already mapped in memory, but cannot be used to retrieve and map the cluster from the storage system).
- SC: Statistical and control information.

e- Regions table(RT).
This is a table that contains entries for each region in the cluster. It is used to keep information about the different regions, so as to help in reorganization, page prefetching, or garbage collection. Each entry contains fields as:

- Loc: Start location of the region relative to the cluster.
- Size: The size of the region.
- SC: Statistical and control information.

3.4 Working With Clusters

3.4.1 Object Creation

Objects are created as a result of invoking the create operation on the associated implementation. The creation environment must be defined in the environment record associated with the context. It must define the logical container (LC), cluster(CID), and region(R). If the creation environment is not initialized when creating an object, the system will assume a default environment to initialize the record. The system default could be the environment of the invoking object, the environment of the invokee (the implementation), or could be a new environment if it is not possible to use the previous two defaults. The user can control the creation environment by either creating a new environment or using an existing one. The primitives used to define the creation environment are:

new(LC) The operation creates a new cluster in the given LC.
The environment record reflects this new environment.

same.as(PID) The operation updates the environment record with the environment of the given PID.

After defining the creation environment, a free space to accommodate the object in the creation cluster is located and a free entry in the private table of the creation cluster is found and used for the object header. The header is then initialized with the implementation identifier, location of the object in the cluster and the object size. The GN and the Mig.flag are set to nil. If the object identifier is to be returned to an object in the same cluster, then the subscript to the created header in the private table is returned as the LID of the object in the cluster. The created object will live in the cluster as a private object (private to the cluster). If the LID has to be passed out of the cluster to another cluster then the object must be made public first. A PID must be requested from the logical container; the GN field in the object header must be updated; an entry in the export table of the creation cluster must be set, another entry in the import table of the referencing cluster must be set and the global name of the object is returned to the referencing cluster.

3.4.2 Object Migration

Objects created in a cluster might be moved to another cluster for different reasons (this will be explained in more detail later in section 4). It is assumed here that only public objects are moved, but we will see later that private objects could move in connection with public objects. The problem with moving an object from one cluster to another is mainly the cost of maintaining the consistency of the references to and from the moving object. The object entry in the storage system must be updated to point to the new cluster.

In the old cluster, all the local references to the moving object must be converted to global references, since it is going to be located in another cluster. An entry in the import table must be set for the moving object, and all private objects referenced by the moving object must be made public and exported via the export table.

In the new cluster, the moving object is recreated as a public object, so it must be given an entry in the export table as well. Objects referenced by the moving object (exported in the old cluster) must be imported in the new cluster.

In the moving object, any references to objects in the old cluster must be adjusted to the PID's of these objects.

It is clear that the operations associated with the migration of an object between clusters could be very expensive. Thus it could be better to delay some of the operations; either to be done later off-line or to be done incrementally. Local references to the moving object in the old cluster can be adjusted incrementally, by keeping the moving object header in the old cluster as a forwarding reference (it will be flagged as a migrating object so the provided global name can be used to update the
Migration operation mentioned previously is that it considers migration between clusters. The problem with the rudimentary port/export tables can be merged to form one cluster. Objects which are not heavily used in their clusters but are highly used objects that are not accessible should be collected. Divisions of independent subclusters can be split into a number of new clusters. Clusters strongly connected through the implementation of such an algorithm requires a slight modification to the mark and sweep or copy and compact algorithms of garbage collection.

4.1 Management Primitives and Implementation

The basic operation of the management primitives is object migration between clusters. The problem with the rudimentary migration operation mentioned previously is that it considers migrating only a single public object, and this would involve a large and unnecessary overhead. If we reconsider the problem more carefully and refer to the points considered in cluster design and in particular the locality formed by dependent components one notes that:

- Objects selected for migration might have some dependent components which should not be left behind nor should they be made public. They should migrate with their host object. This could save a lot of unnecessary work.

- There could be some natural subclusters within a cluster. The subcluster would consist of a set of public objects together with a set of private objects. This set of private objects can be designated as dependent components on the associated set of public objects. These two points were reconsidered in the revised version of object migration. In the following, a list of the management primitives is given, together with a brief description of each primitive.

migrate.L(List, PID) Effect: migrate the list of public objects together with their associated private objects to the cluster containing the object identified by PID, and do the necessary updates. The design decision taken in implementing the migration primitive, is to move a set of public objects together with their associated dependent private objects, instead of just a single object. The problem then became how to find the set of private objects to be migrated and what are the updates required to keep the integrity of references. This is done as follows:

- Given the list of public objects to be migrated from a cluster, find their reachable subset of local objects in the cluster.

- Find the intersection (IS) of this reachable subset (S1) with the remaining subset of local objects reached by the remaining public objects in the cluster (S2).

- Objects in the (IS) referenced by S2 have to be made public.

- Lastly migrants (public and private objects in S1) are physically moved to the selected destination cluster.

The implementation of such an algorithm requires a slight modification to the mark and sweep or copy and compact algorithms of garbage collection.

- A request to migrate objects could be handled by traversing the graphs rooted at the migrating roots, and marking all the reachable objects from these roots in the cluster.

- The other graphs rooted at the remaining roots are traversed. If garbage collection is required, then all reached objects are copied to another space in the cluster to perform compaction while doing the garbage collection, otherwise copying is skipped. While traversing the graph, any objects already marked (from the first step), will be made public.

- Marked objects are then moved to the destination cluster.

Assuming a linked list representation of the graphs in the cluster, and assuming K roots, each requires a complexity
of \(E_r + V_r\) to traverse its corresponding graph, where \(E_r\) is the number of edges, \(V_r\) is the number of vertices of the graph rooted at root i, then the complexity of doing the migration, together with garbage collection (if required) is \(\Phi(\sum_{i=1}^{n} (E_r + V_r))\). To dynamically migrate objects, the overhead could be high, unless the highly mobile objects are given special consideration, so that the containing cluster will restrict having but a limited number of public objects in the cluster. The other alternatives is to migrate the whole cluster containing the object, or to migrate only a single object at a time as in section-3.4.2.

**split(List)** Effect: create a new cluster, then migrate the list of public objects to the new cluster.

**archive(List)** Effect: The archival cluster associated with the cluster containing the list (of public objects) is used as the destination for migrating the List. In this case no object reached by any remaining object in the cluster is migrated. Thus objects reached by the remaining roots are marked, and the (non marked) objects reached by the idle roots are moved.

**merge(PID1, PID2)** Effect: merge the two clusters containing these two objects in one cluster (as a convention the result cluster will be that containing PID2). This operation can be implemented by migrating all the objects having entries in the export table together with their transitive closure of private objects in the cluster containing PID1 to the cluster containing PID2.

**garbage.collect(C)** C is a cluster name. The operation is used by the system, to perform the off-line garbage collection. Thus migrating the list of all public objects together with their transitive closure to a new cluster will create a garbage free cluster.

5 Object Data Management System (ODMS)

Objects in a database may be accessed in a way somewhat different from the way other objects are accessed. In programming environments objects are manipulated through object invocation using the selected operation to manipulate the object state. In most implementations object properties are clustered together and the object state is stored in a contiguous format which provides an efficient access to the object in memory, using a pointer to the object and an offset to each property. In database applications objects do not need to be always accessed directly by name. Instead, access to objects which satisfy certain predicates based on their properties is required. So there is a need to be able to scan through these properties, over a finite closure of objects, before reaching the target objects.

To do that efficiently, properties from different objects need to be clustered together, rather than clustering object's properties together. In addition, some auxiliary structures may be defined to enhance the search efficiency. The objective is to provide efficient searching, over a minimum amount of data blocks in the storage, and also within the least subset of the objects in the objects closure, that would satisfy the query. These object database requirements have been supported by providing an object type called a 'class'.

In relational databases, a relation consists of a finite number of objects (tuples), all having the same structure. A relation could be fragmented horizontally, and/or vertically and could be implemented by a number of files. The class also compiles a finite number of objects, of the same type, but not necessarily of the same structure, since we have allowed for type variants. The class could be fragmented by considering more than one criteria.

Objects of similar structure can be grouped together in a class called Intermediate-Class. Objects in an intermediate-class could be fragmented (horizontally), into what is called later subclasses, and the subclass could be fragmented vertically, into storage segments, each segment comprising, for each object in the subclass, a subset of the object properties.

5.1 Classes In The ODMS

The special ODMS requirements have been achieved using an object called a 'class'. A Class in the ODMS is defined for a particular object type. All the instances of the class must be confined to the definition of their type. The class provides the proper implementation(s) for its instances which implement the behaviour specified by the instance type (interface). The class may also provide some other interfaces as described below.

5.1.1 Class Interfaces

**Query Interface.** A query to the class (a predicate defined on any subset of the class properties) is in fact an operation invocation on the class. The result of the query is an object which could be a collection (or even a class) of objects fragments selected from the invoked (queried) class. The query is analysed (regarding the properties indicated in the query) by the class, and then handed to the proper subset of its intermediate classes that can handle the query (as will be seen later in section-5.2.1).

**Invocation Interface.** This interface is the same as the one provided by the type. The class must provide at least one implementation of the invocation interface. The implementation could be based on the object state as perceived at the class level, or it could be based on the contiguous representation of the object state. The first alternative requires an interaction with the query interface to obtain the object properties required for the invocation, while the second alternative requires a check-out of the object from the data base before invoking the operation. The two alternatives would differ in the efficiency.

**Exchange Interface.** This is provided to handle the movement of objects in and out of a class. There are basically two operations associated with the interface:

- **Check.in(PID, Class)**. An external object named 'PID' is brought into the class 'Class'.
- **Check.out(PID, Imp)**. An object named PID is moved out from its class then rebuilt (recomposed), according to the supplied implementation Imp. If Imp is not provided the object will be recomposed according
to the default contiguous implementation associated with the class.

Management Interface. The interface provides operations to define the criteria for allocating the objects in the class (allocation schema) to the different clusters in the system and defining the storage structures (storage schema) for the objects in these clusters. The interface also provides operations to monitor the high level activities of the class (queries) for the purpose of adapting the different schemas (allocation and storage) to the varying query patterns.

5.2 Class Organization

With reference to (fig.3), one can define the following blocks.

5.2.1 Class-Objects

The class-object provides the high level interfaces listed above. It has an active data dictionary, containing information about the different properties specified in the type associated with the class. Allocation criteria and storage criteria for the objects in the class are also contained in the class-object. Depending on the number of variants associated with the type, there is an equivalent number of intermediate-class objects. Each has an active access to the allocation and storage criteria of the class. When an object is created or checked-in into the class, the object state is matched to one of the type variants, and an intermediate-class is selected. The instances belonging to an intermediate-class are further allocated to different clusters according to some allocation scheme associated with the intermediate-class. The allocation criteria could be based on the values of the different fields of the object state, possibly the creation environment, and some other criteria. To have the allocation fully carried out the object must have enough instantiation (the properties considered in the allocation criteria must be given values) otherwise the object creation will follow some default measures regarding allocation. As an example to check-in an object into a class, the type of the object is checked. The object state is also checked to find the type variant of the object, then an intermediate-class is selected and invoked to carry out the allocation procedure. This either selects the corresponding subclass, or creates a new subclass for the matching allocation criteria. The object is then migrated to the cluster containing the subclass object, and the subclass is then invoked to proceed with the relevant operation.

5.2.2 Subclasses

The subclass contains a subset of the objects, which belong to the corresponding intermediate-class (it can be compared with a horizontal fragment of a relation). To allow for a reasonable amount of independence, each subclass has a copy of a subset of the dictionary contents which concern the operations of its objects, (eg. for each operation, the dictionary may contain information about the fields involved in the operation, and the status of these fields: being read only, update, etc.). The operation dictionary will support invocations based on the query and update operations of the class. It minimizes the overhead of the invocation, by restricting the storage access to the least number of segments required by the invocation. The subclass also contains a storage schema which is passed to it by the intermediate-class. An object allocated to a subclass will be vertically fragmented (decomposed) and stored at the different segments of the subclass. The subclass is also considered the link between the objects it stores and the run time component of the system. Invocations on the ODMS objects are handled by the subclass either using the contiguous state, (the object is temporarily checked-out, invoked, then checked-in back), or using the decomposed state. In either case the proper implementation has to be selected. The subclass as an object, is confined to one cluster together with all the object headers of its members (note that the member objects states are not necessarily in the same cluster).

5.2.3 Segments

These are the lower level objects that contain the actual states of the objects in the class (compared with files). Each segment is implemented as a collection, containing a subset of the object fields together with the object identifier (local identifier) which is used as a surrogate to join the different parts of the object state in the different segments. The segment maintains its state by providing operations to add and remove objects fragments. It also does the search for the objects fragments satisfying the given conditions based on their attribute values or identifiers.

6 Integrating The ODMS In The Overall System

As seen above, the class can be considered as a stand-alone object data management system for its own objects. It has a basic query interface, where the query parameters are checked (in the class-object). Simple search optimization can be done by restricting the search space. This starts by selecting a subset of the intermediate classes that can handle the query (based on the given parameters), each selected intermediate-class selects a subset of its subclasses (based on the allocation conditions and perhaps some other meta information), and finally each subclass selects the proper segments, that can satisfy the query (based on the storage schema). The question then is: Are the objects in the ODMS compatible with the rest of the objects in the object space? The answer is yes. From the behaviour point of view, objects are the same, the difference could be only in the implementation of the object interface and the object structure. ODMS objects are structured to provide for efficient handling of queries, and that may not be the best structure for object invocation. On the other hand objects in the object space are structured to provide for efficient invocation. In some applications, objects to be invoked may not be known until some queries are made to first find these objects, and then operations are invoked on them. This requires that ODMS objects should be handled efficiently from both query and invocation perspective. The efficiency of the object invocation in the ODMS has been handled by providing the class with alternative implementations. One implementation acts on the decomposed state of the object (as represented in the ODMS), and the other operates on the contiguous state of the object. On the other hand, classes have been provided with an exchange interface, so that objects can move freely between the ODMS space and the object space.
Fig. 3 - Class structure in the ODMS
The user can choose the way that best suits his requirement. He can explicitly check an object in or out of the ODMS, or he can leave it to the system default measures. The ODMS also can be provided with the knowledge to decide which implementation to chose for an object invocation so that objects invocation can be made transparent and efficient.

6.1 Object invocation revisited

To offer a uniform invocation mechanism to all objects in the system, the new version of the procedure used to invoke objects, can be summarized by the following steps, which refer to the steps (a, b, c) listed in section-2.

a1- In step (a) the object is tested for being an ODMS object. If it is an ODMS object then step b1 is followed, otherwise proceed with step b.

b1- The header of the ODMS object does not refer to the object implementation but instead, it refers to the subclass. The subclass in turn refers to the two alternative implementations. One of these implementations is selected.

c1- If the selected implementation was the decomposed state version, then the operation is carried out using the decomposed state. If the implementation selected was the contiguous version, then the object must be recomposed using the check out operation on the subclass. Step (c) above can be carried out, then on return from the invocation the object must be checked in into the class.

7 Conclusions and Future Work

A model of an object oriented system has been described, and fully implemented using C-Prolog. The main features that would support efficient operation, as well as simpler management of the system have been emphasized. The system is currently operational. Objects in Prolog can be created as instances of Prolog coded implementations (preprocessed form), or database classes. Objects in the object store exist in clusters, but can move between clusters to form closer localities. Clusters need not be known to users, but can be considered as system entities, which can be observed and controlled by some management tools.

There was a noticeable improvement of object invocation when the invoked objects were clustered. An experiment was made to invoke a complex object, comprising six components, and their associated three implementations (code). Initially the object was split over four clusters, then gradually merged into one region in one cluster. The time of invocation was measured as follows:

Clustering condition          Invocation time(seconds)
- Four clusters               0.533
- One cluster, four regions   0.231
- One cluster, one region     0.183

Note that, due to the fact that the system is implemented in C-Prolog, the figures indicated cannot be considered as performance figures. They can be used only as a base of comparison, specific to this implementation.

ODMS classes provide a fine tuning for the object locality, by considering the internal state of the objects, and providing a high level interface for management, where the allocation and decomposition of objects is controlled by an easily handled allocation and decomposition schemata.

The system has been provided with clear measurement points, that can be monitored (not discussed in this paper). The next step planned is to make use of the monitoring information as well as an expert knowledge and rules, so that the system can be self controlled, regarding the distribution of objects, locality, and objects decomposition and clustering.

Using C-Prolog, it was noticed that the concept of clusters, if supported in Prolog implementation, could facilitate providing a support for efficient management of large object oriented databases in Prolog. In the current implementation of the system the C-Prolog heap was used to process large numbers of objects, with an overall size much larger than the available heap. This was due to the heap management at the cluster level, where only subset of the clusters needed was kept in memory while the rest is unmapped (transparently, similar to paging). The mapping and unmapping in the current implementation is not efficient, because the heap in C-Prolog is not really partitioned, thus mapping and unmapping clusters relied on assertion and retraction.

Acknowledgment

The authors would like to thank all the members of the DSG group at Trinity College, Dublin, who made this work possible. We will not name them individually, but a particular thanks must go to Neville Harris, the group leader, for his encouragement and support. We would also express our appreciation to all the colleagues in our ESPRIT COMANDOS consortium.

References


[16] J. Rothnie, T. Lozano, Attribute based file organisation in a paged memory environment, CACM v 17 n 2, Feb 1974


[19] James W. Stamos, Static grouping of small objects to enhance performance of a paged virtual memory, ACM trans. on computer systems, v2, N 2, May 1984


[22] Ifor Williams, Mario Wolczko, Trevor Hopkins, Dynamic grouping in an object oriented virtual memory hierarchy, Proc. of European conference on object oriented programming, 1987


190