Orthogonal Persistence in a Heterogeneous Distributed Object-Oriented Environment

Pedro Sousa, André Zúquete, Nuno Neves and José Alves Marques

email:{pms,avz,nuno,jam}@inesc.pt
INESC — IST. Rua Alves Redol n°9, 1000 Lisboa, Portugal

This paper describes the major issues in the design and implementation of orthogonal persistence in IK. A single and uniform programming paradigm is used to manipulate objects in a persistent and distributed environment. Object references can be freely passed during remote invocations or stored persistently. IK supports orthogonal persistence with type inheritance. Objects are stored persistently when reachable from an Eternal Root, regardless of their type. For programmers, objects are created and manipulated uniformly, independently of the time they persist.

Persistent objects are dynamically grouped at run-time into clusters to encapsulate fine-grain language level objects into coarser-grain entities. We present a novel approach to integrate object clustering, naming and garbage collection in persistent systems, and present experimental results.

1. INTRODUCTION

Object orientation have been explored in many projects to handle distribution (Jul *et al.*, 1988; Black & Artsy, 1989) and persistence (Richardson & Carey, 1989; Chase *et al.*, 1989) of fine-grained objects. We feel these two dimensions should be addressed together providing distribution and persistency of fine-grained objects in a uniform model, where object references can be freely passed in remote invocations or stored persistently.

IK (Marques & Guedes, 1989; Sousa *et al.*, 1993) is an object-oriented platform which intends to simplify the development of distributed and persistent applications. IK follows the general architecture and model defined in the Esprit project Comandos (Marques *et al.*, 1988; Cahill *et al.*, 1993). It generalizes the view of volatile and persistent data by considering all objects to be maintained by the system while being part of the transitive closure of an Eternal Root. From the programmer's point of view objects are created and manipulated uniformly, independently of their type and the length in time they persist. This paper highlights the key points in the design and implementation of orthogonal persistence, and presents some results from experience.

IK is mainly targeted to cooperative applications executing in a local network comprising a few tens of workstations under a common administration policy. Applications are written in EC++ (Sequeira & Marques, 1991; Sousa *et al.*, 1993), a language syntactically similar to C++ but with some semantic extensions and restrictions. The programming environment is based on a version of the ET++ (Gamma *et al.*, 1988) library ported to our platform, providing a framework for application development. The current version of the system runs in user mode on a network of heterogeneous Unix machines – Suns (3 and SPARCstations with SunOS/Solaris), Bull DPX/2 (B.O.S.) and i386 based machines running the Mach 3.0 micro-kernel (Castro *et al.*, 1993).

The next section overviews the major decisions concerning the persistence model and system design. Then, we present the related work in section 3. In section 4 we discuss the naming mechanisms used in IK. Clustering techniques are presented in section 5, and, in section 6, we explain the implementation of the persistent store. Garbage collection is addressed in section 7. Finally, we draw our conclusions in section 8.

2. MAJOR DECISIONS

2.1. The Persistence Model

We followed the concept of orthogonal persistence, as described by Atkinson (Atkinson *et al.*, 1983). Three principles underly the concept of orthogonal persistence:

- *Persistence independence.* Object persistence is independent of the way objects are created and manipulated. Object I/O is transparent to the programmer.
- *Type orthogonality.* Any object can be persistent regardless of its type, including code.

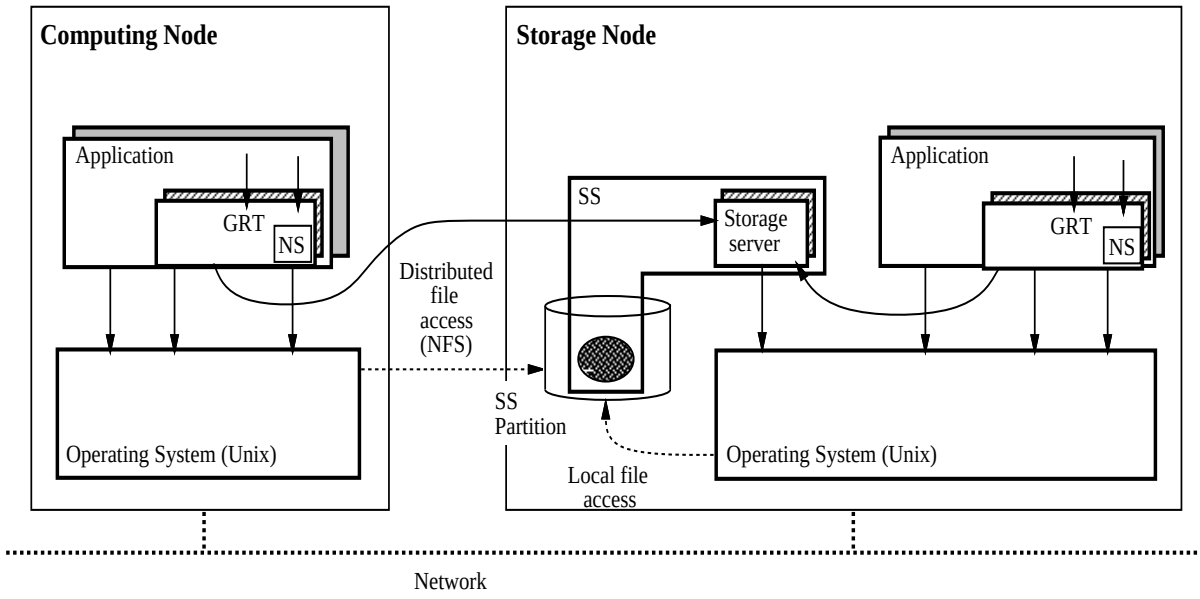

FIGURE 1. Simplified architecture of IK.

- *Persistence identification.* The mechanism to distinguish persistent objects from non-persistent ones is independent of the type system and computational model.

In IK all objects are created and accessed in a single and uniform manner. Persistence is a dynamic attribute of all objects, and orthogonal to their types. Objects are maintained by the system while reachable (either directly or indirectly) from an Eternal Root. Objects are automatically discarded when no longer reachable from this root.

Programmers do not need to know *which* objects are actually persistent, nor *when* they are retrieved or discarded, nor *how* object data is actually stored. Persistency is controlled simply by registering the roots of object graphs in the Eternal Root.

A fundamental aspect concerns with the visibility of object clustering policies at the programming level. One may argue about the flexibility of user defined grouping policies, where programmers can specify if two objects should be placed in the same cluster or in different ones. With orthogonal persistence, programmers often do not know exactly which objects will be stored, and consequently, in which cluster they should belong to. Therefore, we chose to hide the assignment of objects to clusters from the programmer. The system first decides which objects go to which clusters and then where objects should be placed within the corresponding cluster.

2.2. System Architecture

In our model every object can be referenced by every other object in the system uniformly. Such feature requires, at least at the architectural level, a global object space where objects are identified with global, unique and long lived names.

The figure 1 shows a simplified diagram of the architecture of IK. The persistent store* is supported by a set of Storage Servers (SS). A generic run-time library (GRT) linked with each application cooperates with the persistent store to support the global object space. We distinguish two types of nodes in the system: computing nodes and storage nodes. The former may have low availability, while the later must be highly available. Applications can execute in any node, but storage servers only run in storage nodes.

The GRT handles objects within a process address space. It includes, among others, a remote invocation mechanism, a garbage collector of volatile objects, a dynamic linker, an object clustering mechanism and an upcall mechanism to invoke methods in a language independent manner.

The platform also includes a Name Service (NS) to assign and de-assign user-defined names (readable character strings) to global names, thus allowing users to refer to stored object graph with friendly names. The collection of <*user-name, global name*> tuples constitute the Eternal Root.

2.3. Design Issues

The first major issue was related to the support of the global object space defined in the architecture. A virtual address space shared by all applications could be used to implement such object space, as in Amber (Chase *et al.*, 1989). We decided otherwise because the 32 bit architectures of existing machines is clearly

*Secondary Storage in the Comandos Architecture.

insufficient to support it, and it would be difficult to implement it among heterogeneous machines. The object space in IK is independent of the virtual address space, and is supported by the persistent store together with the multiple and disjunct applications' address space.

A second major issue was related with the implementation of the persistence model. Although it does not distinguish between volatile and persistent objects, we wanted to explore their different characteristics. Volatile objects are expected to die young and to be known only within the scope of a single application. Persistent objects normally live much longer and are known by many entities in the system. Thus, volatile objects do not require a global name nor a persistent region, whereas, in principle, persistent objects require both. So, we decided to create every object as volatile, and promote them to persistent on demand. Object promotion occurs when a reference is inserted in another persistent object, or when it is sent to other applications during remote invocations.

We also wanted a global name space to allow each object to refer uniformly any other object. However, we do not want to waste global names with persistent objects that are only being referenced by closely related objects. In other words, we want to store objects in the persistent store without having to assign a global name to each one, and still maintain the ability to refer objects uniformly. This lead to a new approach in the definition of clusters. The main idea is to use clusters as a mechanism that encapsulates referenced objects within referencing ones. If encapsulated objects are only referenced from the encapsulating ones, only the top level object requires a global name, and it is the only object "seen" by the persistent store.

Another important decision concerns with the object location mechanism. Rather than keep track of objects, storage servers simply remember the applications to which objects have been delivered. Once an application retrieves an object from a storage server, it becomes "responsible" for that object, namely, it has to save the object back to the persistent store, and proceed with incoming remote invocations to the object. This responsibility ends when the object is stored back to the persistent store, or when the application explicitly delegates the responsibility to another application, by notifying the persistent store. Such approach allows applications to chose the granularity of object migration they are willing to handle, and avoid third party dependencies imposed by the system.

3. RELATED WORK

Most persistent systems fail to offer orthogonal persistence because they distinguish persistent objects on an allocation basis rather than on a reachability basis (Richardson & Carey, 1989; Kim *et al.*, 1989; Andrews *et al.*, 1990). In these systems objects are persistent if created as such, either by being explicitly declared as persistent or by being created in a default persistent region. Therefore, programmers must worry about persistent objects containing references to volatile objects (dangling references). They also lose some flexibility because they cannot decide about the longevity of objects that already exist. Another common limitation of persistent systems (often caused by allocation-based persistence) is to make persistence an exclusive property of some types. Even though programmers can define their own persistent types, they either make all instances of that type persistent or none at all.

Some systems do not distinguish between volatile and persistent objects and simply make the entire address space persistent, as in Casper (Vaughan *et al.*, 1992). Here, the persistent store provides an abstraction similar to a stable distributed shared memory, with all clients sharing the same address space. We promote independence between applications' address space, and thus the global object space must be independent of applications' address space.

Outside the Comandos project, GemStone (Bretl *et al.*, 1989) and Emerald (Black *et al.*, 1986) are probably the systems closer to IK, since they support fine-grained objects uniformly. GemStone provides the extensions to Smalltalk-80 (Goldberg & Robson, 1983) that are needed in an Object-Oriented DBMS, namely: support of a multi-user and persistent environment, data integrity and a large object space. However, the architecture is based on a central entity that manages the whole object space, somewhat similar to Casper's architecture. In these architectures object space is defined as the set of objects known by the persistent store only. In IK, the object space includes the objects in the persistent store and all the other objects that exist within application address spaces. This provides an higher degree of flexibility, because applications can interchange references to objects not kept in the persistent store.

Even though Emerald (Black *et al.*, 1986) does not support orthogonal persistence, it also provides a uniform paradigm and was particularly important in the definition of our computational model. A significant difference is that, in Emerald, types are classified at compile time and all objects of the same type are either global or local. In IK, objects are promoted individually to global objects at run-time. Emerald exploits the call-by-move parameter passing mode. However, much of the work is left to the Emerald compiler as it must decide which objects are passed using the call-by-move mode. We wanted to keep the decisions generic and independent of the language.

Within the Comandos project Guide (Balter *et al.*, 1991), Amadeus (Cahill *et al.*, 1990) and IK share the same model and general architecture, however, they differ both at the language and implementation levels. Guide followed an integrated approach in which a new language and the run-time were developed to-

gether. IK and Amadeus chose C++ as main programming language. However Amadeus add new keywords to the language to express persistence and distribution, whereas we decided to use C++ inheritance and retain the original language syntax. In Guide, all objects are allocated in memory shared by all applications within a node, and invocations can proceed locally. In this approach, the compiler must prevent applications from damage the entire set of shared objects, which is hard to achieve in case of application crashes. Both IK and Amadeus chose to keep applications' address space private and forward invocations between applications.

In Guide, since all objects are created in shared memory, they all receive a global name and become persistent. In IK and Amadeus global names are assigned to objects when their references are sent between applications. IK goes a step further by grouping objects in such a way object graphs cross address spaces under a single global name.

The current version of IK does not implement the full Comandos architecture, in particular it does not support the notion of Atomic objects (Mock *et al.*, 1992) existing in the Amadeus implementation.

4. OBJECT NAMING

This section addresses the different levels of object naming in IK. We first describe how global names are created and managed in the system, and then the default object location mechanism. We then discuss how and when global names are converted to and from language specific object references, and finally we present the implementation of the Eternal Root.

4.1. Global Names

In centralized systems global names can be generated based on local counters. In decentralized systems a solution is to assign a globally unique tag to each node, which can then be combined with locally generated values to create globally unique identifiers (Needham, 1989). The way local and global unique values are combined must also be commonly agreed.

One could use existing communication addresses, such as IP addresses, as globally unique tags for each node, and allow each and every node to generate global names. However, since in most situations global names are only useful with some location information, we decided to use the unique tag also as a hint to the location where the object is most probably found, as in Appolo Domain (Leach *et al.*, 1983). This does not bind the object to any physical location, it simply avoids the use of other hints when the one embedded in the global name is valid.

In a persistent environment, the storage server where an object will eventually be stored is a good location hint, because it remains true until the object migrates to another storage server[†]. This approach allow us to assign globally unique tags only to storage servers, but it requires application's GRT to request global names from the storage servers where objects will eventually be stored. In the current implementation, the choice of a storage server is an administrative issue and depends on the user running the application. The advantages of such architecture are twofold:

- The management and assignment of unique tags is greatly simplified, since only well-known and stable storage nodes are uniquely tagged.
- The tag in objects' global name is a valid location hint as long as objects do not migrate to another storage server, which is expected to be the common case.

If an object migrates from the default server to another storage server, a forward pointer is left behind. To allow physical servers to be replaced or migrated within the network, as often required by performance or administrative reasons, unique tags are logical values whose association with physical servers is variable over time. Bindings between unique tags and storage server addresses are kept in system files and globally accessible through the Unix NIS service.

In the current version of IK, global names are 64 bit wide, and are formed by a 16 bit storage server tag (*SSid*) and a 48 bit local value. Local values are never interpreted outside the storage server where they were generated, allowing each server to implement different algorithms for name generation and object location.

4.2. Object Location

The location algorithm finds objects given their global name. Objects can be found either locally, mapped in the address space of other applications or in the persistent store. We wanted to solve efficiently the common cases, but also avoid the complexity of a complete decentralized algorithms as in (Fowler, 1985; Black & Artsy, 1989), and third party dependencies of centralized solutions. To determine the common cases, we trace early applications and IK releases and found out that:

- An object is mostly found where it was invoked before.
- If a reference for an object was obtained as a parameter in a cross-context invocation, then the location hint existing in the invoking GRT is most probably valid.
- If a reference was retrieved from the persistent store, then the object it refers to is probably also in the persistent store, and not mapped in some application.

[†]We are only concerned with long term validity. Hints may be temporarily invalid when objects are mapped in applications.

The first observation shows that object migration is rare compared with object invocation, and the GRT should cache the location of remotely invoked objects. The second states that it is worthwhile to pass location hints along with global names when these are sent as parameters of remote invocations. Finally, the third observation shows that it is not worth to store persistently location hints (together with global names) to applications where objects are currently mapped.

The location algorithm is a protocol between applications' GRT and storage servers. First the GRT tries to access the object using the cached location hint. Location hints are updated when new ones are received in incoming remote invocations. In the current implementation, if different hints exist for the same object, we merely chose the last one received. We could improve hint hit ratio by associating counters to them, as in (Fowler, 1985).

If the location hint does not exist, or it is obsolete, the GRT asks the storage server bound to the *SSid* tag in the global name for the object. In the common case, the object is in the persistent store and the storage server retrieves it, keeping a handle to the application that received the object. If the object was already mapped by an application, the storage server returns the application's handle instead, and the request is forward to it. The same procedure is carried out when a forwarding pointer to another storage server is found instead.

If the storage server has no knowledge of the object[‡], it forwards the request to the application that allocated its global name. This requires storage servers to remember the global names allocated by each application. In the current implementation this information is minimized by further dividing the 48 bit local value of the global name into a 32 bit application identifier (*SSgn*), and a 16 bit counter (*GRTgn*) left for applications private use. Applications allocate global names in chunks of 2^{16} names, maintaining a reasonable tradeoff between efficiency and usage of global name space.

Applications are free to migrate objects on their own as long as they are able to locate them, because storage servers do not keep track of the current location of objects, but instead the applications that are responsible for them. If the responsible application crashes or does not answer within a bounded time, the system assumes the current object state is lost, and the last committed state is available under warning. This state is either the state the object had before being mapped in the faulty application, or a newer one if it has been explicitly flushed.

Although the location mechanism provided by the system to locate an object is centralized on a given storage server, it allows a potentially high level of optimizations because applications can actually chose the granularity of object movements to be handled by the system. In particular, short-term migration of objects, such as the ones implied in call-by-move (Jul *et al.*, 1988) parameter passing semantics, can be handled by the applications, without system knowledge.

[‡]The most common situation happens when the global name corresponds to an object that has never left the memory of the application where it was created, and therefore has never been stored in the persistent storage.

4.3. Object References

In distributed systems unique identifiers are expensive entities and should be avoided. Whenever possible, cheaper object references should be used instead of glob -al names. The determination of the necessary and sufficient referencing mechanisms in the system is a critical step in its design. In our system we distinguished three different situations where specific reference mechanisms are useful: within a single address space, in the persistent store, and in cross-context invocation messages.

Manipulation of language level references is part of the application's algorithm and has an immediate impact in its performance. Since these references only need to be valid within the applications' address space, virtual pointers can be used to achieve an inexpensive referencing mechanism. In the persistent store it is important to keep references as small as possible to reduce both the overall size of stored objects and the cost of object I/O. Since objects are stored in clusters, references can be small if their scope is restricted to the cluster where they are stored. On the other hand, the size of references passed during remote invocation is not a critical factor, but they must be valid in the entire system.

In IK, language level references are virtual memory pointers to generic object headers. These headers are used to represent local, unmapped or remotely mapped objects. If a language level reference is sent as parameter in a remote invocation, a global name is allocated to the object (if it didn't have one already) and sent instead, along with a location hint. If a reference is stored persistently it is swizzled into a cluster-wide offset.

Pointer swizzling can be implemented in two basic ways. Global identifiers are converted to pointers just before being used by the application (*on demand*) or immediately after being mapped (*on mapping*). Conversion on demand has been used in several systems (Atkinson *et al.*, 1983; Black *et al.*, 1986; Richardson & Carey, 1989), where a software check precedes each pointer dereference. If the pointer is a global identifier, the object, or a surrogate, is mapped and the global identifier is replaced with the local pointer. In contrast, conversion on mapping ensures that only local pointers are seen by applications, avoiding the checks on pointer dereference. A major disadvantage of conversion on mapping is the need to follow pointers extensively during pointer swizzling. This forces the complete mapping of object trees, leading to immediate and unnecessary conversion of references. Such mapping and conversion wave can be bounded to the size of pages (Wilson, 1991), which may still lead to a

significant amount of unnecessary conversions, because pages are much larger than the fine-grained objects we support.

We wanted both to hide global names from application code and to minimize unnecessary conversions between local and global references. Therefore, we decided to convert references on a per object basis, the first time an object is invoked in a context. The strict encapsulation enforced by the model ensures that object instance variables are not seen by the application before the first invocation to the object. Therefore, by trapping the object's first invocation, references can be swizzled and always appear to be in the local format to applications code.

4.4. The Eternal Root

As mentioned in section 2, the Eternal Root is a collection of <*user-name, global name*> tuples. Since we did not want to implement a name server, we use the underlying file system as one. User-defined names are names of the underlying file system. The association between an user-defined name and a global name is also maintained in the file system, by creating an symbolic link from the user-defined name to a (unexisting) file whose name derives from the global name. To later retrieve a global name given the user-defined name, the NS looks for that name in a set of directories, either given explicitly by applications or defined by environment variables. The use of links rather than small databases to hold <user name, global name> pairs allows users to see object names as normal file names in their working directories.

5. OBJECT CLUSTERING

In the Comandos model, all objects reachable from the Eternal Root are maintained by the system. This does not mean that all objects reachable from the Eternal Root have a representation on the persistent store. In fact, reachability from the roots is computed only at some points in time, normally when applications exit or when explicitly demanded.

Since we are dealing with a potentially large number of fine-grained objects, clustering techniques are of great importance because they increase locality of reference by co-locating related objects, reducing the number of page faults and disk I/O. Such technics have been extensively studied in the scope of Object Oriented Data Bases (Tsangaris & Naughton, 1992), and they try to find the best placement for objects within a cluster. However, the assignment of objects to clusters has been left for programmer. As explained in sections 2.1 and 2.3, we decided to assign objects to clusters at run-time, and hide as much as possible object's graph under a single global name.

The assignment of objects to clusters is done in such a way that:

- Clusters have only a single globally known object, called head object.
- The body of the cluster is composed by all objects that are only reachable through the head object, i.e. there are no references outside the cluster for the cluster body, only for its head object.

Therefore, objects belonging to the cluster body do not need a global name because they are only referenced within the cluster, and the whole cluster can be identified by the global name of its head object.

Since there are no references to the objects encapsulated in a cluster, no special addressing mechanism is required to locate these objects. With this approach, the system can benefit from the encapsulation provided by such clusters and consider the whole cluster body as private data of the head object. This organization minimizes the number of globally known entities, allowing an entire object graph to enter or leave address spaces with a single global name. It simplifies garbage collection within clusters, because the head object is the only root to be considered. It also scales down the problem of global garbage collection since clusters are the entities to be recycled.

In IK, objects are clustered when they leave the application's address space, either to be saved in the persistent store or migrated to other applications. A cluster is mapped upon invocation of its head object. When that happens the GRT maps the whole cluster (using a memory-mapped file) and unfolds the head object[§]: converts its data to the current machine format and swizzles its references. The other objects will be unfolded only when invoked. This way we delay object unfold until they are accessed and also defer touching cluster pages until necessary.

5.1. Assigning Objects to Clusters

As mentioned before, at some point in time the application's GRT must detect the objects reachable from the Eternal Root among those existing in their address space and save them in the persistent store. In general it is difficult to known exactly which objects are these, because references to them may have been passed to other applications, and their usage is unknown. Thus, we take a conservative approach and consider that all objects with a global name must be saved. The GRT keeps track of all global names known by the application in a table called globally Known Object Table (KOT).

In a first phase we consider each object in the KOT as the head of a new cluster. The graph of each head object is traversed and visited objects are tagged with the mark of the head object. The traversal does not propagate to objects in the KOT, because they will be the starting point of new traversals. If an object is

[§]Object unfolding is not actually done by the GRT. Instead it upcalls the language specific run time to perform such operation.

tagged with different marks, meaning that it is reachable from different cluster heads, it is promoted to a new cluster head: a global name is assigned to it and inserted in the KOT, to be traversed next. The entries in the KOT that refer to objects already unmapped or mapped remotely are ignored.

After marking all objects reachable from the KOT with a cluster head mark, objects with the same mark are grouped together. The head object is inserted first, and the others are appended to the cluster. Objects are folded as they are being inserted into the cluster: references are swizzled to indexes in the corresponding cluster table and their data is translated into a well known format (XDR: eXtern Data Representation format). There are two tables in a cluster: one holding outgoing references (global names) and another holding offsets of objects within the cluster. Within the cluster object references are indexes of these tables.

Applying the algorithm to the example of figure 2 would result in four clusters. Objects `A` and `B` are globally known and are registered in the KOT. Objects `C` and `D` cannot be hidden in `A` or `B` private subgraphs and are promoted to cluster head objects. The figure also illustrates the cluster created to store object `D` and its private subgraph.

If an object in the KOT is the head of a cluster mapped from the persistent store, than many objects of that cluster may still be on disk, since objects are loaded on demand. Thus, we must consider the pros and cons of rebuilding the cluster from scratch or simply appending new objects to its end. Rebuilding the whole cluster may require the loading of many objects, to save them back again. On the other hand, adding new objects to the cluster is easy, since they can be placed at the end of it. However, this does not recycle garbage objects in the cluster, wasting disk space, spreading live objects among garbage ones and reducing locality of reference. Scanning the whole cluster may prevent us from saving garbage, but requires us to read and write back the whole cluster. Currently the decision to rebuild a cluster is based on the percentage of objects that remain folded. If the number is higher than a threshold percentage, the traversal of the object graph stops whenever it reaches an object still folded. New objects are appended to the cluster, and old ones are saved in their original positions¶. If most of the objects in the cluster are already unfolded, we simply scan the whole graph as when creating new clusters.

¶The decision to append new objects to a cluster produces a positive feedback, since the number of garbage objects in the cluster tends to increase each time the cluster is stored without being traversed, therefore increasing the probability that the same decision is taken on future cluster storages. The cycle is broken when clusters are garbage collected, as explained in section 7.

5.2. Arranging Objects Within Clusters

There are two basic types of algorithms for ordering objects within clusters: trace-based and graph-based. Trace-based algorithms create a weighted graph of objects and references using application traces. Objects with (and related by) higher weights are placed together. Weights can be simply proportional to the number of invocations (Hudson & King, 1989), computed using Markov chains (Tsangaris & Naughton, 1992), or can be tunned using "attraction" and "repulsion" forces (Grourhant *et al.*, 1987). The effort is paid back in disk intensive applications with fairly constant access patterns to the graph of objects in the persistent store, such as in Object Oriented Data Management Systems (Tsangaris & Naughton, 1992). Graph-based algorithms traverse the object graph in a depth-first or breadth-first manner and group objects as they are reached. These algorithms are simpler, and tend to group objects poorly than trace-based, since they only see instantaneous objects relations rather than access patterns. In a system with orthogonal persistence each and every object can potentially be clustered and saved persistently. To use a trace-based approach, one must gather information about invocations on all objects. Since most objects are in fact recycled by the local garbage collector, most of the information gathered would be useless, and the whole process too costly. Therefore, we chose a graph-based approach with the descendants traversal ordering defined at compile time. Graphs are scanned using a depth-first algorithm because of its simplicity and because it showed the best behavior among several graph-based algorithms (Stamos, 1982).

5.3. Considerations on Object Flushing

IK offers object flushing for checkpoint purposes. Applications can flush objects and recover back the flushed state, by explicitly calling the GRT primitives. Upon termination of an application, objects are saved and the flushed state is overwritten. If the application crashes before saving the object, the flushed state becomes automatically available. The clustering algorithm used for object flushing differs from the one used for object saving, because:

- We expect flushing to be a much more frequent operation than recovering, since the first occurs in the due course of the application execution and the second occurs as the result of error recovery.
- Flushed objects remain mapped in the application, and therefore the image sent to disk remains invisible to other applications.

Under these conditions, we decided to simplify the flushing and traverse only the graph of the object being flushed, rather than traverse all objects in the KOT as before. The objects visited during the traversal that are

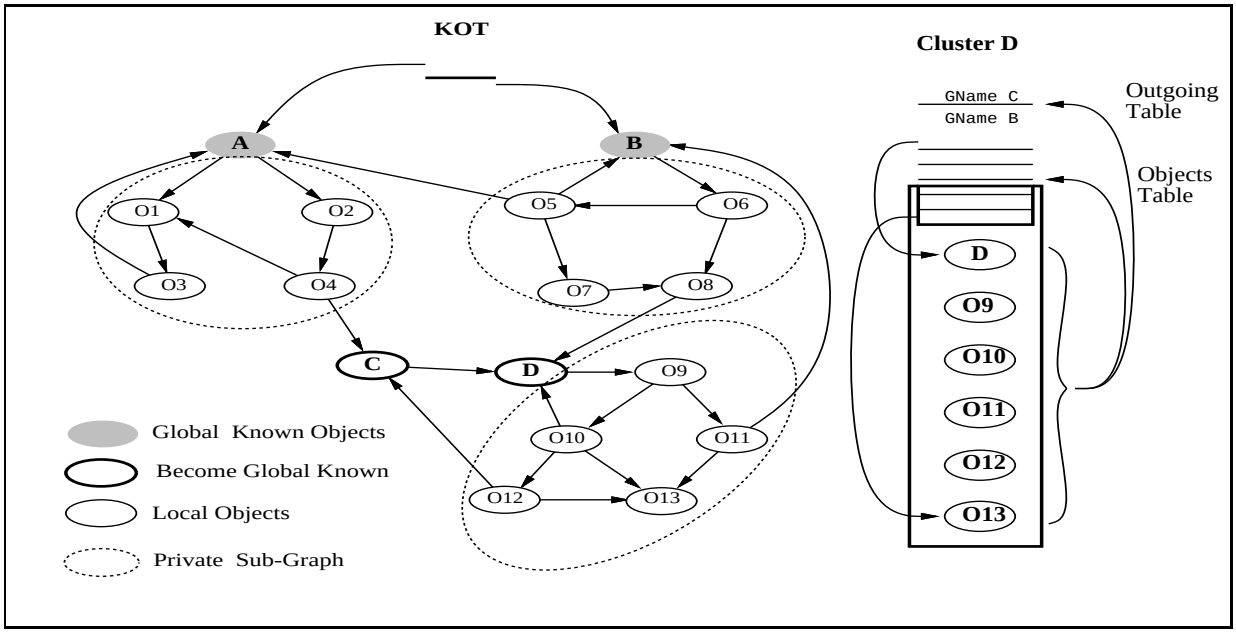

FIGURE 2. Definition of clusters from object graphs.

not in the KOT, are copied‖ into the cluster body. If the graph reaches an object in the KOT, it is recursively flushed. The control returns to the application when all clusters have been successfully flushed.

6. THE PERSISTENT STORE

This section describes the implementation of the persistent store on top of the NFS distributed file system. There are two basic approaches to deal with object storage: direct file access from contexts to databases or plain files, as in (Richardson & Carey, 1989), or interaction with servers responsible for managing centralized or distributed databases, as in (Bretl *et al.*, 1989; Andrews *et al.*, 1990). In IK we adopted a mixed model that provides direct access from applications to object storage (files) and servers to control the aspects related with protection and synchronization. Direct file access optimizes data transfers between the persistent store and the applications, and allows the use of memory-mapped files to lazily and efficiently load cluster pages into applications**. The use of servers allow us to keep object protection independent from file system protections. As applications have direct access to the object storage, we use a simple manipulation of file protections to give temporary access to applications. By default, containers are readable and writable only by the storage server. When an application wants to map a cluster the server changes the protection and group of the respective container, giving read or read/write access to the application. There is a well known group per user to be used in these cases.

Next, we had to decide the way clusters are allocated among files, i.e., if there is one or many per each file. Since we did not know what would be the average size of clusters, we decided to start with the simpler solution: use one file (container) per cluster (or class). Storage servers do not impose any internal structure for containers, they are simply data segments identified by a global name. Containers are protected files, with a name deriving from their global names, and are stored in globally accessible directories (SS Partitions). Global access is achieved using NFS. There is one storage server per SS Partition, and its status is maintained in log files.

Memory-mapped files are used to map clusters into application's address space. However, to ensure the consistency of the original image of clusters in disk, we must prevent the operating system (e.g. Mach) to swap out memory-mapped containers to themselves during the normal execution of applications. Therefore, modified blocks must be explicitly written using the usual UNIX I/O interface.

Normally, applications update the data of a container overwriting its contents. As this operation is not atomic, a cluster may become inconsistent if the application crashes while saving the it. We decided to support also atomic update of individual clusters. Basically, we relay on the "atomic" update of UNIX directories to update clusters atomically. Before storing a cluster an application requests the storage server to create a new container. After updating its contents on disk, the application requests the storage server to substitute the old container by the new one. Future accesses to the cluster use the new container.

‖During the copy, references are swizzled, the data converted to XDR format and the object is tagged with the mark of the cluster head object.

**Also possible using servers as external pagers like in Mach based operating systems as in (Vaughan *et al.*, 1992).

7. GARBAGE COLLECTION

To simplify the problem of garbage collecting objects in a persistent and distributed system, we use three independent collectors: one to recycle objects within an application's address space, a second one to recycle objects within a cluster, and a third one to eliminate clusters that are no longer reachable from the Eternal Root.

Local objects are recycled using a generation scavenging algorithm (Ferreira, 1991). In our environment, for local collections to proceed autonomously, it is also necessary to consider the entries KOT as local roots. When a cluster is mapped, clustered objects are considered to belong to the oldest generation.

Cluster-wide garbage collection recycles dead objects within each cluster. As was explained in section 5, some clusters are stored without being fully traversed and may containg garbage. Since the cluster head is the only globally known object in the cluster, clusters can be garbage collected autonomously: any object in the cluster not reachable from the head object is certainly garbage. Clusters are garbage collected off-line, by a process running continuously on each storage node. Clusters are locked while are being recycled, forcing applications to wait until the cluster is saved back.

Finally, a system-wide garbage collector deletes clusters in the persistent store that are no longer reachable from the Eternal Root. Currently, this collector is a straightforward implementation of the mark-and-sweep algorithm. It is comprehensive, centralized, non-robust and the whole system must stop for garbage collection. It is mentioned here because it had an important role in the conclusions.

8. EVALUATION AND CONCLUSIONS

8.1. The Persistence Model

Orthogonal persistence provides an high level of transparency, hiding from programmers all non-relevant aspects. The usual question is to what extent can this be done without sacrificing other attributes, particularly performance. In systems where data must be stored explicitly, programmers tend to store only the long-term information (i.e. required to survive between application runs); no transient information is stored. With orthogonal persistence, selection is based on reachability criteria, and programmers must keep this in mind during the design of their applications. They should be aware that:

- Objects should contain either long-term or transient information.
- Objects with long-term information should not refer to objects with transient information. They should be kept in different graphs, otherwise the system can unnecessarily save a potentially large amount of transient information.

The first point is not an issue in well designed applications. In fact, our measurements have shown that only 2% of the instance data of objects containing long-term information is transient. We are thus led to the conclusion that good application structuring tends to separate persistent and transient information in different objects.

The second point has no simple solution since graphs with long-term information often need to refer to graphs with transient information. We use upcalls to solve this problem. The GRT upcalls the *beforeUnmap* method on an object before traversing its references. Through the redefinition of this method, programmers can clear the references to transient graphs, while long-term graphs are saved. Similarly, the GRT upcalls the *afterMap* after having unfolded an object, allowing the programmer to reinitialize the references to transient information.

A different source of problems, not directly related to the model but rather to the implementation, occurs when programmers use language level object references (virtual memory pointers) as numeric values, for example, to implement hash functions. Since swizzling global names into local references yields different values on different application runs, the contents of a language level reference changes between application runs. This prevents programmers from using language level references for other purposes than pointer dereference and equality tests. Offering a generic primitive to return the global name of an object was not a feasible solution because most objects never become globally known, and so no global name is assigned to them.

8.2. Object Clustering

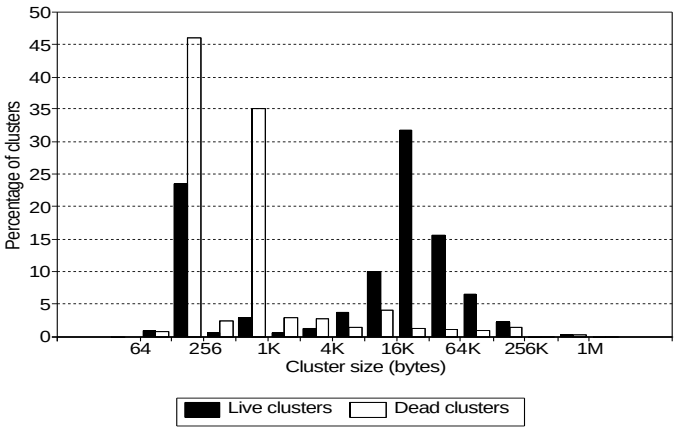

FIGURE 3. Distribution of Cluster Population.

The evaluation of the clustering mechanism is based on the applications built on top of IK. Some of the applications are a cooperative calendar manager that allows several users to schedule and negotiate meetings, a cooperative graphical editor that supports multiple users

concurrently, a browser and an inspector for application development and a tool for the analysis and design of distributed applications. The applications and the ET++ library all together comprise 452 classes and 6216 methods, a number we consider sufficient for a minimal evaluation of the platform. The average size of objects in clusters is approximately 42 bytes.

In figure 3 we present the distribution of cluster population according to their size. Bars represent the percentage of clusters whose size is between two consecutive powers of 2. White bars refer to garbage clusters and dark ones refer to clusters reachable from the Eternal Root. Approximately 70% of alive clusters are larger than 4K bytes, most of them containing between 500 and 1000 of objects. They result from applications that handle information in a hierarchical fashion, where nodes of the graph are not exported outside the graph. The remaining 30% of alive clusters consist of very small graphs containing only a few objects. Typically, such clusters are generated when graphs are traversed remotely. In fact, some of our applications do tend to fragment clusters completely, because they scan lists mapped remotely, forcing the system to promote each visited element to a cluster. This is probably the main drawback of our clustering approach. However, such applications tend to perform poorly given the large number of cross-context invocations. Some of these applications have been changed to improve performance, mainly by reducing the amount of cross-context invocations. They were redesigned to hold local replicas of frequently accessed objects, being able to compute with a reduced number of remote invocations. The reduction of clustering fragmentation was noticeable.

Analysis of the population of dead clusters has revealed that about 80% of the clusters are smaller than 1K bytes. They contain mainly transient information used during the communication between the various components of distributed applications. In most cases, such objects become globally known when references to them are sent as parameters in remote invocations. Our conservative approach, in considering that all references sent to other contexts are reachable from the Eternal Root, promotes these objects to cluster heads and saves them, together with their private subgraphs, when the application finishes. We implemented a simple termination protocol between the various components to detect and ignore such garbage. However, we found out that most of the garbage generated in this way forms distributed cycles, not easily collectible. We are now planing to migrate such clusters to a unique component, which would be able to recycle them more easily.

An important aspect of the evaluation is to find out how clustering actually reduces the number of globally known entities in the system. This is given by the ratio between the number of clusters and the number of objects, which is approximately 0.005. This means that, in average only 1 object out of 200 that is stored in the persistent store becomes globally known. These numbers encourage us to design better location and migration mechanisms to improve migration of single objects between storage servers, since the necessary tables would not be too large.

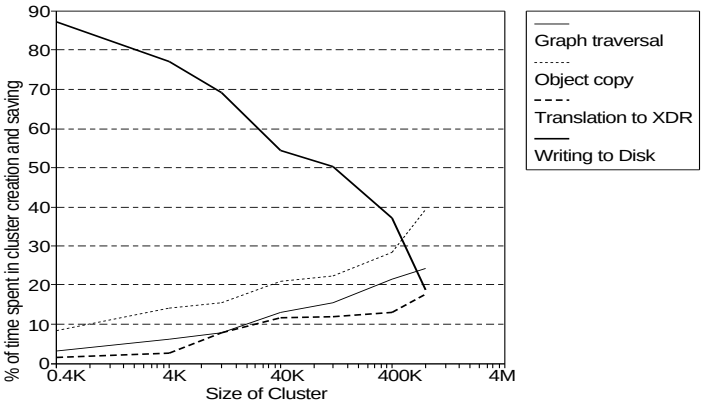

FIGURE 4. Costs of Cluster Creation

A complete evaluation of the costs of clustering mechanisms would require an exhaustive trace in a representative set of applications. We measured the costs of object clustering only as a function of the cluster size. We traced the clustering of object graphs during the execution of our applications, rather than use a single well known graph. Figure 4 presents the relative cost of each phase of cluster creation. One can see that, for clusters smaller than 40K bytes (the majority of them), graph traversal uses less than 13% of the total time to create and save a cluster.

The results presented in figures 3 and 4 clearly show the need for some storage strategy to handle small clusters more efficiently, rather than store them in individual files. We are extending the SS to use simple key-value databases to store small clusters, and send their data along in messages exchanged between applications and storage servers.

To conclude, although the proposed clustering strategy is not a complete solution, it is a valuable component that scales down the cardinality of object population. It highlights relevant objects and hides the others. It can be used as a filter in traditional key-value databases easing the management of large population of objects.

Acknowledgments. We would like to thank Paulo Guedes, Manuel Sequeira, António Rito, Paulo Ferreira, Cristina Lopes, José Pereira, João Pereira, David Matos, Helena Oliveira, Pedro Trancoso and Miguel Castro for their work on the IK system.

References

Andrews, Tim, Harris, Craig, & Duhl, Joshua. 1990. *The Ontos Object Database.*

Atkinson, M. P., Bailey, P. J., Cockshott, P. J., Chisholm, K. J., & Morrison, R. 1983. An Approach to Persistent Programming. *Computer journal*, **26**(4), 360–365.

Balter, R., Bernadat, J., Decouchant, D., Duda, A., A. Freyssinet, S. Krakowiak, Meysembourg, M., Dot, P. Le, Nguyen, H. Van, Paire, E., Riveill, M., Roisin, C., de Pina, X. Rousset, Scioville, R., & Vandome, G. 1991. Architecture of Guide, an Object—Oriented Distributed Operating System. *Computing systems*, **4**(1), 31–67.

Black, A., Hutchinson, N., Jul, E., & Levy, H. 1986 (September). Object Structure in the Emerald System. *Pages 78–86 of: Oopsla '86 proceedings.*

Black, Andrew P., & Artsy, Yeshaymu. 1989 (June). Implementing Location Independent Invocation. *Pages 550–559 of: Proc. of the 9th int. conf. on distributed computing systems.*

Bretl, Robert, Maier, David, Otis, Allen, Penney, Jason, Schuchardt, Bruce, Stein, Jacob, Williams, E. Harold, & Williams, Monty. 1989. The GemStone data Management System. *Pages 283–308 of:* Kim, Won, & Lochovsky, Frederick H. (eds), *Object oriented concepts, databases, and applications.* ACM press, Frontier Series.

Cahill, V., Balter, R., Harris, N.R., & de Pina (Eds.), X. Rousset. 1993. *The Comandos Distributed Application Platform.* Springer-Verlag.

Cahill, Vinny, Horn, Chris, Kramer, Andre, Martin, Maurice, & Starovic, Gradimir. 1990. C** and Eiffel**: Languages for Distribution and Persistence. *In: Osf microkernel applications workshop.*

Castro, Miguel, Neves, Nuno, Trancoso, Pedro, & Sousa, Pedro. 1993 (April). MIKE: a Distributed Object-oriented Programming Platform on top of the Mach Micro-Kernel. *In: Proceedings of the usenix mach conference.*

Chase, J., Lazowska, E., Amador, F., Levy, H., & R.Littlefield. 1989 (December). The Amber System: Parallel programming on a network of multiprocessors. *In: Proc. of the 12th acm symp. on operating systems principles.*

Ferreira, Paulo. 1991 (October). Reclaiming Storage in an Object Oriented Platform Supporting Extended C++ and Objective-C Applications. *In: Proc. of the int. workshop on object orientation in operating systems - ieee.*

Fowler, R. J. 1985 (December). *Decentralized Object Finding Using Forwarding Addresses.* Ph.D. thesis, University of Washington, Seatle, USA.

Gamma, Erich, Weinand, André, & Marty, Rudolf. 1988 (October). ET++ - An Object-Oriented Application Framework in C++. *In: Euug.*

Goldberg, A., & Robson, D. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

Grourhant, Yvon, Louboutin, Sylvain, Cahill, Vinny, Condon, Andrew, Starovic, Gradimir, & Tangney, Brendan. 1987 (4-8th October). Dynamic clustering in an object-oriented distributed system. *In: OLDA-II (Objects in Large Distributed Applications).*

Hudson, Scott E., & King, Roger. 1989. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *Acm transactions on data base systems*, September.

Jul, Eric, Levy, Henry, Hutchinson, Norman, & Black, Andrew. 1988. Fine-Grained Mobility in the Emerald System. *Acm transactions on computer systems*, **6**(1), 109–133.

Kim, Won, Ballou, Nat, Chou, Hong-Tai, & Garza, Jorge F. 1989. Features of the ORION Object-Oriented Database. *Pages 251–282 of:* Kim, Won, & Lochovsky, Frederick H. (eds), *Object oriented concepts, databases, and applications.* ACM press, Frontier Series.

Leach, Paul J., Levine, Paul H., Douros, Bryan P., lton, James A. Hami, Nelson, David L., & Stumpf, Bernard L. 1983. The Arquitecture of an Integrated Local Network. *Ieee journal on selected areas in communications*, **1**(5).

Marques, José Alves, & Guedes, Paulo. 1989 (2-6th Octer). Extending the Operating System to Support and Object-Oriented Environment. *In: Proc. of the oopsla 89.*

Marques, José Alves, Balter, Roland, Cahill, Vinny, Guedes, Paulo, Harris, Neville, Horn, Chris, Krakoviak, Sacha, Kramer, Andre, Slattery, John, & Vandôme, Gerard. 1988. Implementing the COMANDOS Architecture. *In: Proc. of esprit technical week.* Brussels, Belgium: North-Holland.

Mock, Michael, Kroeger, Reinhold, & Cahill, Vinny. 1992. Implementing Atomic Objects with the RelaX Transaction Facility. *Computing systems*, **5**(3), 259–304.

Needham, R. M. 1989. Names. *Pages 89–101 of:* Mullender, Sape (ed), *Distributed systems.* ACM press, Frontier Series.

Richardson, Joel E., & Carey, Michael J. 1989. Persistence in the E language: Issues and Implementation. *Software - practice and experience*, **19**(12), 1115–1150.

Sequeira, Manuel, & Marques, José Alves. 1991 (October). Can C++ be Used for Programming Distributed and Persistent Objects? *In: Proc. of the int. workshop on object orientation in operating systems - ieee.*

Sousa, Pedro, Sequeira, Manuel, Zúquete, André, Ferreira, Paulo, Lopes, Cristina, Pereira, José, Guedes, Paulo, & Marques, José Alves. 1993. Distribution and Persistence in the IK Platform:

Overview and Evaluation. *Usenix computing systems*, **6**(4), 391–424.

Stamos, James W. 1982 (May). *A Large object-oriented virtual memory: Grouping strategies, measurements, and performance.* Tech. rept. SCG-82-2. Xerox PARC, Palo Alto, California, USA.

Tsangaris, Manolis M., & Naughton, Jeffrey F. 1992 (May). *On Performance of Object Clustering Techniques.* Tech. rept. 1090-1992. University of Wisconsin-Madison, USA.

Vaughan, F., Basso, T., Dearle, A., Marlin, C., & Barter, C. 1992. Casper: a Cached Architecture Supporting Persistence. *Computing systems*, **5**(3), 337–359.

Wilson, Paul R. 1991 (October 17-18). Operating System Support for Small Objects. *In: Proc. of int. workshop on object orientation in operating systems.*