Object grouping in EOS

Olivier Gruber, Laurent Amsaleg

▶ **To cite this version:**

Olivier Gruber, Laurent Amsaleg. Object grouping in EOS. [Research Report] RR-1686, INRIA. 1992. inria-00076910

HAL Id: inria-00076910
https://inria.hal.science/inria-00076910

Submitted on 29 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1686

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

OBJECT GROUPING IN EOS

Olivier GRUBER
Laurent AMSALEG

Mai 1992

RR.1686

Object Grouping in Eos

Olivier Gruber, Laurent Amsaleg
Projet Rodin
INRIA, Rocquencourt
78153 Le Chesnay Cedex, France

{gruber, amsaleg}@rodin.inria.fr

Abstract

Eos is an environment for building distributed object-based systems. Leos, the language for Eos, provides transparency for distribution and persistence. In this paper, we address the problem of declustering the object graph onto a number of nodes and of locally clustering objects within pages with minimal impact on the programming process. We propose a grouping model which on the one hand achieves full transparency. The grouping is dynamically achieved by the run-time system as directed by user-provided hints. This dynamic object grouping copes automatically with evolutions of the object graph. The implementation incurs little overhead since it is a side-effect of garbage collection. On the other hand, our model supplies Eos users with an explicit and fine control over data and computation placement so they can load balance the overall system.

Mots Clés : productivité, transparence, distribution, placement d'objets, placement de traitements, localité de référence, dynamicité du placement

Le Groupement d'Objets dans Eos

Eos est un environnement destiné à la construction de systèmes distribués à objets. Le langage d'interface à Eos, dénomé Leos, fournit une complète transparence à la distribution et à la persistance des objets. Dans ce papier, nous proposons un modèle de placement miminisant l'impact sur le processus de programmation à la fois du partitionnement du graphe d'objets sur un ensemble de nœuds et du regroupement d'objets dans des pages locales à un nœud. Notre modèle assure d'une part une complète transparence au partitionnement du graphe. En effet, le placement des objets est dynamiquement assuré par le système en fonction d'indications définies par les utilisateurs. La dynamicité de ce mécanisme permet la réorganisation automatique du placement des objets en présence d'évolutions de la topologie du graphe d'objets. Ce mécanisme est mis en œuvre comme un effet de bord du glanneur de cellules auquel il n'ajoute qu'un faible surcoût. D'autre part, les utilisateurs d'Eos conservent un contrôle explicite sur le placement des données et des traitements afin d'équilibrer globalement le système.

Object Grouping in Eos

Olivier Gruber, Laurent Amsaleg
Projet Rodin
INRIA, Rocquencourt
78153 Le Chesnay Cedex, France

{gruber, amsaleg}@rodin.inria.fr

Abstract

Eos is an environment for building distributed object-based systems. Leos, the language for Eos, provides transparency for distribution and persistence. In this paper, we address the problem of declustering the object graph onto a number of nodes and of locally clustering objects within pages with minimal impact on the programming process. We propose a grouping model which on the one hand achieves full transparency. The grouping is dynamically achieved by the run-time system as directed by user-provided hints. This dynamic object grouping copes automatically with evolutions of the object graph. The implementation incurs little overhead since it is a side-effect of garbage collection. On the other hand, our model supplies Eos users with an explicit and fine control over data and computation placement so they can load balance the overall system.

1 Introduction

The Eos system [11] is a general-purpose programming environment for building multi-user softwares that need sharing, persistence and distribution capabilities. In this context, productivity of programmers is a prime goal. Research and industry both strive to deliver general-purpose programming plateforms which increase productivity by reducing programming overhead. Overhead comes from a lack of transparency in the following aspects.

- Persistence.
- Distribution.
- Grouping.

The lack of persistence transparency used to force programmers to write persistent specific code. Programming languages following the persistence model of [1] provide such transparency. In this model, persistence is orthogonal to type, i.e., persistence is not provided through type extents. Second, persistence is orthogonal to instanciation, i.e., persistence is not specified when

objects are instanciated. Third, persistence propagates automatically to all objects reachable from a system-defined root of persistence.

The lack of distribution transparency used to force programmers to manually locate their data and explicitly ship either computations or data. Distributed object-oriented programming languages provide the ability to invoke an object without knowing its actual location.

The lack of grouping transparency used to force programmers to write specific code to group objects with two constraints. First, they have to decluster (or partition) the object graph over nodes. Second, they have to further cluster objects locally within pages. Existing persistent systems provide transparency based on built-in collection constructors, e.g., the relation constructor in relational database systems. These built-in constructors provides physical independency of user programs. However, these systems are not extensible since new collection constructors cannot be added.

The Eos programming environment provides both persistence and distribution transparency. The language of Eos, named Leos, is the interface of the store. Leos is an object-oriented programming language which is made generic through parametrized classes [14]. The root of persistence is a global naming service, called the catalog. Distribution transparency is supported by the system-wide store. The store follows a Non-Uniform Memory Access model (NUMA) [4, 5] over a distributed architecture, i.e., the store is fragmented over a collection of nodes which are interconnected by a local area network. Each node has processing power and fast access to its local storage, and slower access to other nodes local storages.

Grouping transparency with respect to pages is upheld, i.e., the store provides the abstraction of a virtual memory heap of objects. The clustering of objects within pages is achieved automatically by the run-time system based on the object composition, i.e., child objects are grouped together with their parent. In case of multiple parents, the upper-levels are required to solve the ambiguity. This is achieved by introducing a relevance label upon composition links. The labelling is performed within classes. Ambiguity is solved by insuring that the highest relevance wins. Transparency is upheld for the programmer because the relevance setting is made in classes thereby making it a schema issue and not a programming issue.

Grouping transparency with respect to nodes is only partially supported. Declustering an object graph first involves a partitioning of the graph into subgraphs which then have to be placed among nodes. While transparency is upheld with respect to the graph partitioning, subgraphs placement is explicitly handled by users. User control is necessary by load balancing because data declustering and computation placement are tightly related and application-dependent. This hurts transparency somewhat but this necessary as there is not yet an execution model which provides a fully automated approach to load balancing.

Eos provides a reachibility-based model to partition the object graph which yields the same degree of transparency as the persistence model. The basic idea is to introduce user-defined roots of declustering. Each root defines a partition of the graph and also constitutes the user handle to place that partition on a node. A partition defined by a root contains objects which are reachable from that root. Link relevances are again used to solve the declustering ambiguity introduced by objects reachable from multiple roots.

Eos' approach to object clustering has several contributions. First, Eos does not trade productivity nor extensibility for performance. Productivity is preserved because clustering is no longer

a programming issue but a schema issue. Extensibility is allowed because our clustering mechanism is dynamic and copes with add-on bulk-types. Add-on bulk-types are programmed in Leos as advocated by [6]. Indeed, its dynamicity enables to support evolutive data structures while existing systems typically adopt a static clustering mechanism. Such dynamicity is implemented as a side-effect of the garbage collection process therefore adding very little overhead.

This paper is structured as follows. Section 2 introduces the Eos data and execution models. Section 3 presents the underlying machinery behind dynamic clustering of objects. In particular, clustering algorithms are presented. Section 4 presents how the garbage collection is the angular stone of our solution and ensures the performance of our dynamic scheme. Finally, we conclude in Section 5.

2 Eos Models

This section first introduces the Eos data model which enable object creation and complex object composition. This model also enables Eos users to indicate the desired partitioning of the object graph for node placement. Next, the execution model is presented which permits Eos users to achieve load balancing by placing on nodes both subgraphs and treatments.

2.1 Data Model

This section presents the Eos data model. A first part of the model matches that of generic object-oriented programming languages. It includes the concepts of base types, classes, and objects. Base types are the classical ones (integer, reals, etc) plus the object reference. A class is a tuple constructor parameterized by base types and a behavior (methods). An object is an instance of a class and has an identifier. A reference holds an object identity and enables to access the corresponding instance.

The second part of the model matches the definition of subgraphs. A first definition of a subgraph is all objects reachable from a given object identified as the subgraph root. Subgraph roots are user-defined and declared by adding a property named *rooted* on objects. This property can be set or removed at any time on any object. Objects which have the rooted property are called *rooted objects.*

Of course, some objects might be reachable from several rooted objects. Hence, such objects belong to more than one subgraphs according to the previous definition. However, our purpose is to produce object subgraphs that can be placed on nodes. Therefore, since objects are not ubiquitous, an object should belong to one and only one subgraph. Without further knowledge, the run-time system can only come up with a non-deterministic choice. This may be perfectly adequate but upper levels may also have enough semantics to advise a better decision.

In order to support a clustering behavior which is deterministic, we introduce a labelling of composition links which indicates their relative relevance. Relevances are integer values which introduce a non-strict ordering between the links pointing to an object. Hence, an object is reachable from one and only one rooted object through most relevant links. Most relevant links are the link with the highest relevance among the links leading to an object. An adequate setting of relevance values is under the upper level responsability. The subgraphs of reachable objects through most

relevant links from a rooted object is called a **Rooted Object Graph** (ROG).

However, an approach in which labels are set per link has one major drawback: transparency is lost. Moreover, the labelling process may prove tedious if each link has to be labelled individually. Finally, labelling each link is costly since it induces a space overhead per link.

The solution to these problems is to move to relevance setting in the schema and more precisely within class definitions. In other words, classes encapsulate the clustering behavior of objects. This approach renews with transparency since the relevance setting is now a schema issue and no longer a programming issue. Moreover, the storage overhead is greatly reduced since the relevance values are stored only once in the class. Note that objects in the store were already aware of their class for several other purposes like late binding or garbage collection and therefore no specific overhead is introduced.

Link relevances are expressed in the **Class Composition Graph** (CCG). The CCG is a graph where vertices are classes and edges materialize object composition. There is a distinct edge from a node (class) X to a node (class) Y if an instance of the class X can reference an instance of the class Y. The CCG represents all the possible compositions of objects based on class definition (including inheritance) and not the actual state of the object graph in the store.

The CCG can be easily expressed in the schema. The notion of class is extended to include a list of possible parent classes. This list is automatically deduced by the Eos schema manager. For a class C, the list of possible parent classes are those whose instances may reference an instance of class C. Relevances are specified within that list for each possible parent class. If no relevance value is provided by the user, a system default value is used which is smaller than any other values.

An example of a CCG is shown in Figure 1. The schema contains three classes: Person, Dog and Kennel. In addition, it holds two parameterized classes: Btree[T] and Hash[T], which are generic collection constructors. These constructors have been instantiated as follows: Btree[Person], Btree[Dog], Hash[Kennel].

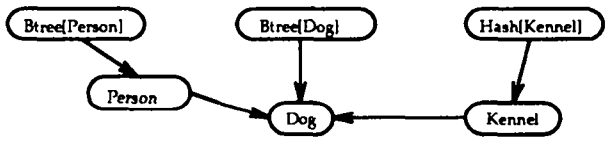

Figure 1: Class Composition Graph and Link Relevance

The corresponding physical schema is made of three named collections: People, Hospital and Kennels. People is the collection of known persons that may or may not own a dog. Kennels is the collection of known kennels in the country. Hospital is the collection of dogs currently undergoing care in the veterinary hospital.

An example of relevance setting for the edges leading to class Dog and the resulting link labelling on the object composition graph are illustrated in Figure 2. The desired clustering is the following. A dog is either owned by a kennel or a person and should be grouped with its owner. Persons win over kennels. Some dogs are ill, and are referenced from the file of the veterinary hospital, but are never clustered by hospital.

The Eos approach to clustering not only provides transparency but also dynamicity. This is

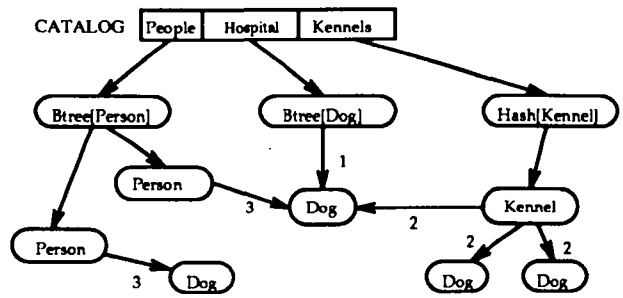

Figure 2: Link Relevance Setting and Object Graph

a prime advantage uniquely supported by our system. The clustering of objects is done asynchronously in background mode by the run-time system. Periodically, the correctness of the actual clustering is checked with respect to the actual state of both the object graph and the relevance setting. This is achieved with very little overhead as shown in Section 3.5. Our approach copes with the following dynamicity.

- Changes in the setting of relevances.
- Topological evolution of the object graph.

If the clustering of some objects is detected as incorrect, they are moved accordingly to the new state to correct the situation. This reorganization is achieved while the system runs. In other words, there is no need to halt the entire system or part of it for reorganizations to take place.

2.2 Execution Model

To enable the programmer to place both subgraphs and threads on nodes, nodes should be made visible in the language and the programmer should then be provided with the two following mechanisms of placement.

- Node anchoring of rooted objects.
- Node targetting of methods.

Nodes are logically designated in Eos through the use of a system table which hides physical nodes, hence physical independence is achieved this way. Beside their specific identity, nodes appear as normal language objects, instances of a built-in system class. The class node answers several default methods as the one providing current load information for example. There are as many instances as physical nodes in the system.

The anchoring mechanism enables to place a ROG on a node. Any rooted object can be anchored at any time at any node. The node of anchoring is specified via the identifier of the node.

A rooted object can be anchored at one and only one node at a given time. However, the anchoring can be dynamically changed. The anchoring of the rooted objects concerns the entire ROG.

The other binding tool is the node targetting of methods. The place of execution of a method is by default the node of the caller, following a data shipping paradigm. In order to support function shipping, methods can be specified a running location.

There are two ways to specify the running location of a method. A first straightforward way is to ship it to its receiver object. By receiver, we mean the traditional notion of receiver of a method, i.e., the object on which method selection is done. Hence, we extend the notion of targetting to the shipping location. One can note that this approach provides full location transparency since our scheme does not require from the programmer a locating phase prior to the calling.

The other way to place a method is explicit, i.e., by providing a node identifier. The use of an explicit node placement enables a finer control over load balancing but does not ensure transparency. Typically, this facility will be used by optimizer-generated codes.

3 Clustering Machinery

A clustering mechanism suggests a notion of physical containers in which objects should be grouped. The rationale for clustering is to obtain a better locality of reference within those physical containers. Thereby, it reduces the negative effect on performance of the small bandwidth and high latency of both disks and networks. We discuss Eos physical containers in Section 3.1.

Object clustering is based on the composition of objects. Objects are clustered within physical containers according to the following rule: child objects near their parent. We believe that object composition is a good foundation for object clustering because it approximates the access patterns of applications since they navigate through the object graph.

When an object is shared, link relevances are used in order to solve the clustering ambiguity. Link relevances enables Eos to determine the set of most relevance parents for an object. *Most relevant parents* are the parents which have the highest relevance. Objects are grouped with one of its most relevant parent. The choice is non deterministic. Note that this non determinism can be avoided if users provide strictly ordered relevances in the schema.

The allocation of objects in physical containers must respect the most relevant parent semantic. Hence, object clustering has three aspects. First, objects should be allocated in the same physical container as their most relevant parent. In particular, we show that our solution is better than traditional ones in case of container overflow. Object allocation is covered in Section 3.2.

Second, the clustering of objects should be dynamically maintained in order to maximize locality if the most relevant parent of an object changes due to a new setting of relevances or a graph evolution. In other words, the store should be able to reorganize itself dynamically. This is done asynchronously in background mode, i.e., in parallel with user programs. This process is presented in Section 3.5.

Third, the internal fragmentation[1] problem due to wasted space in physical containers is discussed. The control of the internal fragmentation level is important for the accuracy of the object

[1] This should not be confused with the declustering or data-partitionning of a collection.

clustering. Otherwise, the amount of wasted space per page increases which in turn increases page faults. Eos adopts an asynchronous packing philosophy which we present in Section 3.4.

3.1 Eos Physical Containers

Physical containers are the harbor and the pier containers. The *harbors* aim at improving node locality. Recall that node locality is indicated by upper-levels via rooted graphs. A harbor is the storage structure of a single rooted graph. Note that a harbor is always local to a single node, independent of whether the root is anchored or not. Conversely, a node may contain multiple harbors.

A harbor is itself divided in pier containers. A *pier* is a set of disk tracks. The size of piers or harbors is limited by the storage capacity of the underlying physical node. The rationale for piers is the following. First, they improve disk bandwidth since transfers take place on a track basis. Furthermore, global disk locality is ensured by clustering related tracks in close cylinders. The track relationship is given by the membership of tracks in piers and piers in harbors. Another purpose of piers is to provide a finer clustering of objects than with harbors.

3.2 Object Allocation

The allocation scheme of objects has a goal and a constraint. The goal is to group correctly the allocated objects. The constraint is the necessary allocation speed in the context of a programming language [10].

An object should be allocated near its first parent. The general rule of the most relevant parent specializes into the first parent upon allocation since there is no other parent. The first parent of an object is perfectly known since we use an Eiffel-like instanciation mechanism, i.e., an object is instanciated by sending a create message to a class-typed attribute of another object.

Most persistent object systems reject an immediate clustering of objects at allocation time. In contrast, they advocate for clustering at commit time for several reasons. First, they claim immediate clustering is too slow. Second, they argue that the clustering at commit time is more effective. Third, they highlight that a computation produces a lot of garbage while garbage collecting the persistent space is costly. Finally, they state that aborts are more expensive because they produce garbage in the persistent space.

None of these reasons is still valid in our environment. First, our mechanism is efficient as shown below. Second, most objects are not shared, and therefore the clustering at creation time will be correct. Furthermore, this avoids an extra copy at commit time and permits the creator to benefit from correct clustering. Third, a computation produces garbage but only for transient objects since their scope is the computation. Fourth, commits are hopefully more likely than aborts.

Thus, Eos adopts an immediate clustering of objects at allocation time. We said that a new object is allocated near its first parent. Ideally, *near* means within the same I/O granule, i.e., the same track. However, this is in conflict with allocation speed. A track is a fixed size entity which may overflow. Hence, the allocation mechanism should correct the overflow situation. There are two solutions. First, the track may be packed. Second, a place may be searched somewhere up a hierarchy of containers. In Eos, it is the track, then the pier containing the track, and finally the

harbor containing the pier. This is inherently slow. Moreover, this produces poor locality in case of concurrent allocation.

Therefore, we adopt a slightly relaxed notion of near. In Eos, near means in the same pier container. A pier container has a variable size and therefore never overflows. When more space is needed, a new track is allocated to the pier. Therefore, object allocation does not deal with overflow and consequently a fast linear allocation can be used.

Of course, this approach is likely to produce high fragmentation within ever increasing piers. This suggests that pier size should be controlled and fragmentation corrected.

3.3 Pier Container Balancing

The accuracy of a clustering policy means its effectiveness in actually increasing performance, which means increasing the ratio between instructions with and without data faults. This suggests two requirements for piers.

- An average pier size close to an optimum value.
- A dynamic balancing of piers.

Clustering accuracy is enhanced by enlarging piers up to a threshold size. As the size increases, a given pier will hold more and more of unneeded objects even in presence of an ideal clustering policy. Hence, the improvement of accuracy will slow down. Moreover, the larger the pier size is, the more sensitive the system is to the accuracy of clustering. On the one hand, the larger the I/O granule is, the more expensive the fault is, but the greater the improvement of an effective clustering is. On the other hand, with larger I/O granules, cache overflows are more likely as the effectiveness of clustering decreases, which in turn increases data faults.

Therefore, the optimum size of a pier should be a system generation parameter. A sound range of values is from a track up to a cylinder. The challenge is to maintain the pier size close to that optimum since piers are of variable size.

Our solution is to view a harbor as a clustering data structure itself. A good analogy can be made with B^+-trees. A B^+-tree groups its elements in pages according to a key order which represents the access pattern. Leaves split in order to cope with overflows while preserving the ordering of elements. Leaves merge or slide elements in order to avoid fragmentation, again in a way that preserves the ordering of elements. The three interesting points are the following.

- The ordering of elements matching the access pattern.
- Leaves split to maintain the physical ordering in presence of page overflows.
- Leaves slide elements to maintain fragmentation to a low level.

The ordering of objects in a harbor is given by the most relevant parent relationship. This is a non-strict order like in B^+-trees with duplicate keys. Piers split in case of overflow in a way which respects the most relevant parent relationship. Piers slide objects to correct fragmentation also

in a way that respects the most relevant parent relationship. Finally, piers adopt a free-at-empty approach instead of a merge one, i.e., piers are deallocated when they are free of any object. The rationale is algorithmic simplicity.

3.4 Pier Overflow Control

A pier overflows when its size is larger than twice the optimal value. This can be the case for two reasons. First, the size of live objects may be effectively above the threshold. Second, the size of live objects is below, but fragmentation might cause the physical size of the pier to be above. The former case is solved by splitting the pier while the later should be solved by packing. As it will become clear shortly, both cases have a single solution.

Pier splitting introduces two problems. First, there is a response-time problem. If piers split when they overflow, the performance of object allocation may severely degrade. Therefore, an immediate splitting strategy is not practicable. On the contrary, Eos adopts an asynchronous splitting strategy.

The second problem is related to the identity of objects. Eos uses a one-level addressing scheme in a one-level store. One-level store means that the persistent object store is directly mapped into the virtual spaces of Eos users [3, 7]. One-level addressing means that virtual addresses are directly used to support object identity. Therefore, when objects move, they leave a forwarder in place which indicates their new location. These forwarders are later reclaimed by the garbage collector.

Such support of object identity impacts the splitting process of piers. A split implies that some objects are moved and both halves are packed. Hence, object addresses change. So, forwarders have to be left in place. Consequently, the splitting process need to split the old pier (called from-pier) in two new piers (called to-piers) so that forwarders can be left in place. The old pier will be freed when all forwarders will have been reclaimed by the garbage collector.

The pier splitting algorithm is an adaptation of a scavenging algorithm [2, 9]. The split is done with respect to the size and according to the most relevant links in order to minimize the relevance of the cutset. The principle is the following. A to-pier is created. The from-pier is recursively walked through *the most relevant links only* starting from a system-defined root. As side-effect of this walk-through, upon recursion backtrack, each object is copied to the to-pier. After each copy, the to-pier size is checked against the optimum value size. If the to-pier size is larger, a new to-pier is created and the copy process is resumed. The algorithm is given in Figure 3.

Each pier has a hidden root object which references a set of user objects. These pointed-to objects are called pinned objects. The name rationale is they are not subject to sliding (see next section). These pinned objects are automatically deduced as a side-effect of pier splitting. In the algorithm of Figure 3, the to-pier size is tested after each recursive copy of a child[2]. Hence, when the to-pier size is detected too large, the objects pointed to by the current cutset between the to-pier and from-pier constitute the pinned set of objects. Originally, a harbor is created with a single pier which hidden root is initialized with the harbor rooted object. Recall that a harbor is the storage structure for a single rooted graph, therefore it has a single rooted object.

Rooted objects and pinned objects are different. Both are user objects. But, pinned objects are pointed-to by the system-defined hidden roots of piers while rooted objects have been user-added

[2]This is equivalent to test after each copy.

```
object::scavenge(pier_t * to_pier)
{
  low_bound = first_child();
  current = low_bound ;
  while ( current != nil ) {
    scavenge(current);
    current = next_child(current);
    if ( to_pier->size() > optimum_value ) {
        // Pin all objects belonging to the cutset
        // as the pinned objects of the current to_pier.
        ...
        // Then allocate a new to_pier.
        to_pier = new pier_t;
    }
  }
  self->copy(to_pier);
}
```

Figure 3: Scavenge and Split Algorithm for Piers

the rooted property in the external clustering model.

An example of pier overflow is given in Figure 4. The left-hand side of the figure presents an harbor containing a single pier. The size of this pier is two tracks, hence it overflows. The right-hand side of the figure represents the harbor after the split of the pier. Please note that the effect of a split is based on the most relevant link. For example, the object called **G** is copied into the first to-pier because the relevance between **E** and **G** is higher than the one between **I** and **G**. The first to-pier reach its optimum value when object **B** is copied. Thus, **B** becomes the pinned object of the first to-pier.

Walking the object graph through the most relevant links is quite easy. Each object is type tagged, i.e., knows its type. Hence, when walking through a link to a child, the types of both objects are known. Therefore, the link relevance is known from the two types by looking up the schema information. Moreover, each object remembers the highest relevance of links pointing to it. Given the highest relevance, and knowing the relevance of the link we are walking through, we are able to tell whether we use a most relevant link or not.

We now move on to the other case of pier overflow: fragmentation. It is now clear that it is a special case of the more general splitting one. The same algorithm can be used since it scavenges (copies and packs) the objects, and then eventually splits if needed.

3.5 Dynamic Reclustering

The object sliding between piers is motivated by dynamic reclustering. Basically, evolutions of the object graph may make an object misclustered. An object in a pier is misclustered if it has at least one parent object outside its current pier with a higher relevance than the relevance of its parents inside the pier. A misclustered object should be moved in the pier of its most relevant

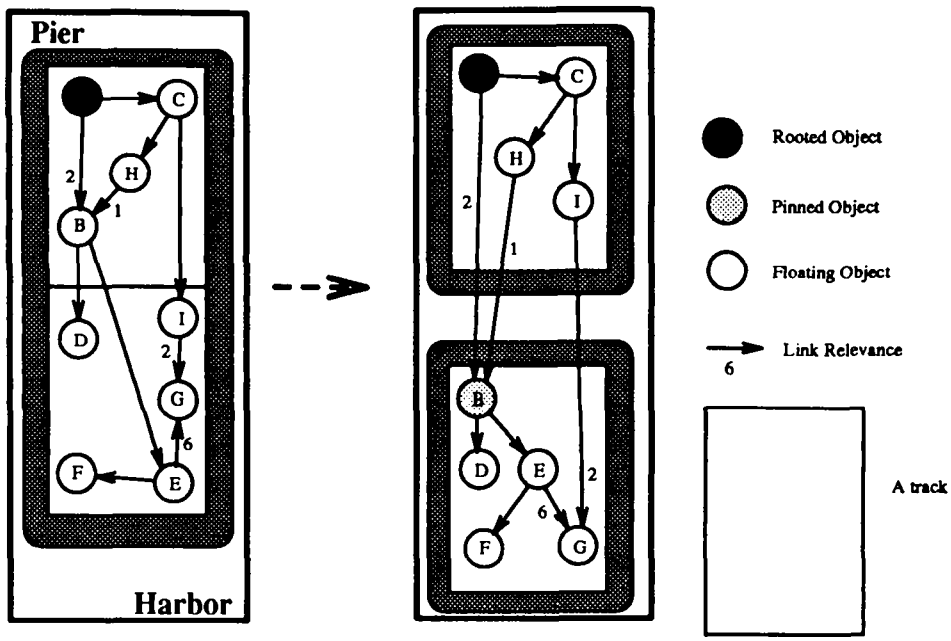

Figure 4: Pier Overflow

parent outside its pier, i.e., the one with the highest relevance.

Please note the dynamic reclustering is stable, i.e., objects having multiple parents with the same relevance should not be alternatively grouped with two or more parents, i.e., crossing a pier boundary back and forth. An object is said to be grouped with another object when it is in the same pier. The reclustering is stable because it respects the following rule.

Rule: An object crosses a pier boundary if and only if an external parent has a strictly stronger relevance.

A special case has to be made with respect to that rule for the pinned objects of a pier. Clearly, these objects will be misclustered since they are at the cutset between two piers, but we consider them as never misclustered since they are pinned.

Object sliding between piers is equivalent to a key update in a B^+-tree. The key provides the ordering and therefore the physical clustering of elements. Hence, when the key of an element is modified, its position in the tree should be modified to match its new key value. In practice, this is often not supported, and a delete/insert approach is necessary.

In fact, sliding an object slides a grape of objects all at once for better performance. The detection of the most relevant parent is asynchronous and done as a side-effect of the garbage collector (see Section 4). Therefore, this knowledge is actualized at a frequency which is too low to cope with an object-based move. Indeed, it would mean an object slide at every garbage collection! Instead of moving a single object, we move its grape. The grape of an object is the set of all objects reachable from that object through the most relevant links and belonging to the same pier. An exemple is shown in Figure 5.

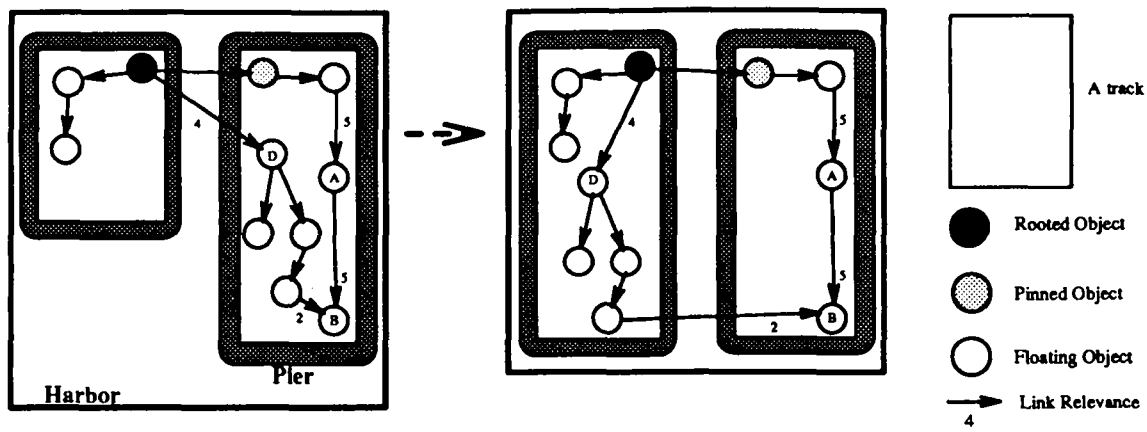

Figure 5: Dynamic Reclustering wrt Link Relevance

3.6 Pier Merge

Piers merge to reduce fragmentation. Fragmentation appears because of garbage, but also because of the previous sliding approach which tends to empty some piers to enhance the clustering of objects.

Pier merge has the same motivations as the leaf merge in B^+-tree. However a B^+-tree merges neighboor leaves. This is feasible in a tree since a tree structure provides a semantics of neighboors. In our store, the ordering is given by the most relevant relationship between objects. So, there is no ordering between piers that can be used to support the neighboor concept.

Therefore, merge of piers is achieved by emptying piers which underflow. A pier underflows when it size becomes under the optimal value. This is done by extending the previous object sliding. The basic idea is to unpin the roots of a pier which underflows. Then, theses objects will be misclustered and Eos will recluster them somewhere else. The pier is thereby emptied and then freed.

An example is given Figure 6. The pier containing **A**, **B** and **C** in the left hand side of the figure underflows. Moreover, object **C** is misclustered because the relevance between **D** and **C** is higher than the one between **B** and **C**.

Our dynamic clustering mecanism first move **C** from its current pier to the one containing its most relevant parent. Then, the pinned object **A** is unpinned and its grape is reclustered.

4 Dynamic Reclustering Support

The dynamic clustering mechanism of objects is based on detection of the most relevant parent. This detection requires the run-time knowledge of all the parents of a given object and the relevances of their links. The relevances are easily obtained from the schema.

The knowledge of parents is efficiently provided by the garbage collection process. The basic

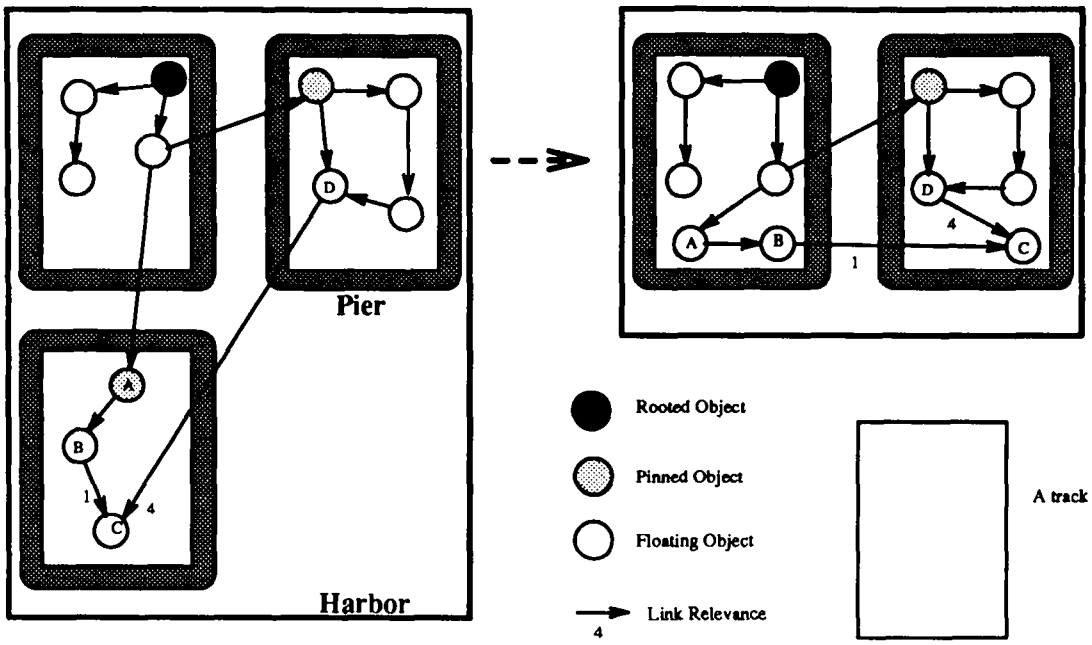

Figure 6: Pier Merge

idea is that an on-the-fly scan-based garbage collector [8] walks through the entire object graph and therefore is aware of all the parents of each object. We point out that this parent knowledge may not be exact if mutators (user programs) run in parallel with the collector and therefore may change the graph topology.

Hence, dynamic clustering adds very little overhead to the garbage collector. First of all, each object remembers the highest relevance between links pointing it. Objects are provided this information by the collector during its walking phase since it views all their parents. Moreover, during its walking phase, the collector detects misclustered objects. I.e., when it walks from a parent to a child, it checks the relevance of that link against the highest relevance stored in the child object. If the link relevance is strictly stronger and the parent and child objects are not in the same pier, then the child object is a candidate to be reclustered. Remark that a candidate object might be discovered as well-clustered because an even stronger parent has been found in its current pier. The objects still candidate at the end of the walking phase are those which are misclustered. Then, Eos is able to move the candidate objects and their grapes accordingly.

Therefore, garbage collector is the angular stone for the performance of the Eos dynamic clustering scheme. But one may argue about the collector very own performance. The rest of this section provides the readers with sufficient insights on the garbage collection in Eos so that they can be convinced of its feasibility.

The garbage collection process has two phases: detection and reclaimation. Both are traditionally glued, but they are achieved independently in Eos.

The detection of garbage is on-the-fly, incremental, and conservative. It is on-the-fly because mutators run concurrently with the collector [8]. It is incremental because harbors are indepen-

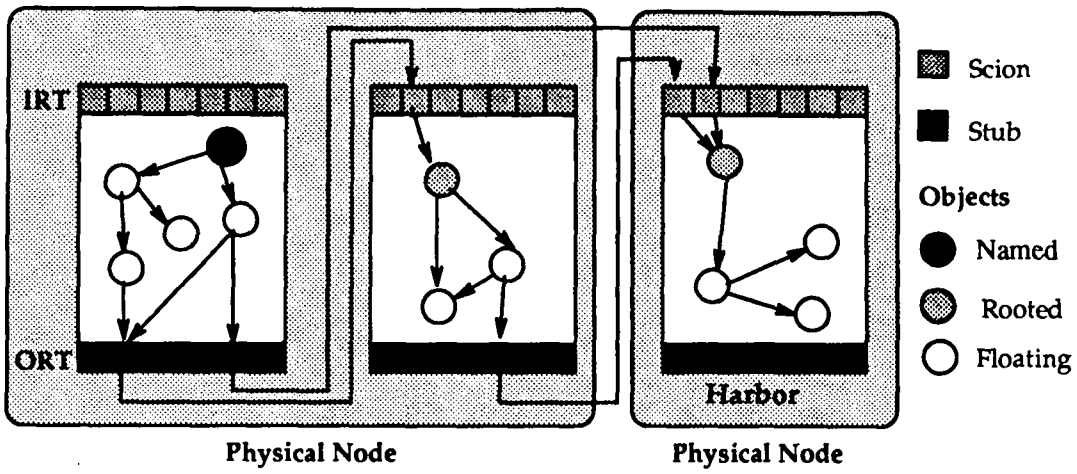

Figure 7: Harbor Isolation Through IRT and ORT Tables

dently collected. It is conservative because each harbor maintains two tables of incoming and outgoing references so that local collections can be independent and without global synchronism [13, 12, 15].

The global sketch is the following. Each harbor is insulated with respect to object composition by two tables. One table called the Incoming Reference Table (IRT) keeps track of incoming references from the outside. Each entry is called a scion. The other table called the Out-going Reference Table (ORT) maintain the outgoing references to objects in other harbors. Each entry is called a stub. For a given object O in an harbor H which is externally referenced, there is a stub in ORTs of harbors having objects which reference the object O and there is a corresponding scion for each stub in the IRT of harbor H. This is depicted in Figure 7.

Each harbor is locally and independently scanned. Local detections are conservative with respect to the IRT, detect local garbage, and therefore clean the ORT. When stubs disappear, the pointed-to IRTs are informed and corresponding scions are in turn removed. The local detections are independent, i.e., there is no global synchronism.

The reclaimation of garbage is separated from its detection. The detection process *detects* garbage and marks the objects as dead at a harbor granule. Then, an asynchronous process incrementally packs the store at a pier granule.

The rationale is that the store is persistent and therefore it is likely that large portions of the store are quite stable from one collecting to another. In other words, only a fraction of the entire store need to be packed after a collecting. An incremental approach to packing is able to benefit from that property while traditional collectors, mark-and-sweep or scavenging, systematically copies the entire store. Note that generational collectors are inherently unadapted to our approach since they are incompatible with a user-driven clustering.

5 Conclusion

Eos is a general-purpose programming environment for building applications that need persistence and distribution capabilities. Such applications are fragile with respect to locality. Locality suggests an adequate grouping of objects in order to improve performance. But grouping control is a major programming burden of existing systems. This paper presented the grouping model of Eos which upholds an almost complete transparency for programmers to significantly reduce this burden.

Eos users are required to provide two kinds of information to the run-time system. First, some objects have to be identified as rooted objects. Second, relevance information need be provided between the potential compositions of classes. In other words, if a class is potentially composed by some others, then the relative importance with respect to grouping of these different compositions have to be expressed in the schema.

The identification of rooted objects and the relevance setting serve two purposes. First, they define a partitioning of the object graph into subgraphs which are meaningful for Eos users, meaningful in the sense that they represent subgraphs in which applications have a good locality. The second purpose is to permit the run-time system to perform entirely the page-level clustering of objects.

This automated page clustering is based on the composition of objects and has several advantages. Objects are clustered within pages according to the rule: child objects near their parent. Link relevances being again used to solve multiple parent ambiguity. We believe that object composition is a good foundation for object clustering because it approximates the access patterns of applications since they navigate through the object graph. The advantages are multiple. First, the clustering is transparent and therefore ensures physical independence of programs. Second, it is dynamic and therefore copes with evolutions of the object graph. Finally, complex reorganizations of the store are simply obtained through relevance changes in the schema; no programming at all is involved in that process.

Moreover, programmers avoid the overhead of the object graph partitioning over nodes, while retaining full control over load balancing. Eos provides a grouping model which supplies the same degree of transparency with respect to this partitioning issue as the persistence model of [1]. Based on the information of rooted objects and link relevances, the object graph is automatically partitioned into subgraphs by the run-time system. These subgraphs are placed onto nodes by Eos users by a simple anchoring of rooted objects. This mechanism associated with the targetting of methods which controls their execution location permits a full control to Eos users over the load balancing of the overall system.

Future work in Eos will strive to provide full transparency of object grouping using link relevances with explicit high-level parallel constructs.

Acknowledgements We are especially grateful to Patrick Valduriez for helping us clarifying this paper. We appreciate the fruitful discussions with Laurent Daynès and Eric Amiel. We also thank Véronique Benzaken for her help.

References

[1] M. Atkinson, P. Bailey, K. Chrisholm, K. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), 1983.

[2] H. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21, April 1978.

[3] A. Bensoussan, C. Clingen, and R. Daley. The Multics Virtual Memory: Concepts and Design. *CACM*, 15(5), 1972.

[4] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple but effective techniques for NUMA memory management. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pages 19–31, Litchfield Park, Arizona USA, December 1989. ACM.

[5] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA policies and their relation to memory architecture. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA (USA), April 1991.

[6] M. Carey, D. DeWitt, J. Richarson, and E. Shekita. Object and file management in the EXODUS extensible database system. *In Proc. of the 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan*, August 1986.

[7] G. Copeland, M. Franklin, and G. Weikum. Uniform Object Management. In *Proc. of the Int. Conf. on Extended Database Technology*, Venice, Italy, 1990.

[8] E. Dijkstra, L. Lamport, A. Martin, and C. Scholten. On-the-fly garbage collection : an exercise in cooperation. *Communications of the ACM*, 21(11), November 1978.

[9] R. Fenichel and J. Yochelson. A LISP garbage collector using serial secondary storage. *Communications of the ACM*, 12, November 1969.

[10] P. Gautron. Experience in programming with C++ parameterized types. Technical Report LITP 90.45 RXF, Rank Xerox, Laboratoire Informatique Théorique et Programmation, Université Paris VI, Paris (France), April 1990.

[11] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez. EOS, An Environment for Object-Based Systems. In *Proc. of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 757–768, January 1992.

[12] J. Hughes. A distributed garbage collection algorithm. *Language and Computer Architecture*, 1985.

[13] R. Ladin and B. Liskov. Highly-available distributed services and fault-tolerant distributed garbage collection. *In Proc. of the 5th Symp. on the Principles of Distributed Computing*, August 1986.

[14] B. Meyer. *Object-Oriented Software Construction.* Computer Science. Prentice Hall, 1988.

[15] M. Shapiro, O. Gruber, and D. Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Technical Report INRIA-1320, INRIA, Rocquencourt (France), November 1990.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399