OISIN: Operating System Support for Objects in a Distributed Environment

Vinny Cahill, Andre Kramer

Distributed Systems Group,
Department of Computer Science,
Trinity College,
University of Dublin,
Dublin 2,
Ireland.

vjcahill@cs.tcd.ie

July 23, 1992

1 Introduction

As part of the Esprit-I COMANDOS project the distributed systems group in Trinity have designed and implemented *Oisin* [2] [3], a native distributed operating system kernel supporting the object oriented style of programming. Oisin currently runs on both bare MicroVax II and NS32032 based workstations. Oisin provides transparent access to both local and remote objects which may be either recently created or long-lived persistent objects (stored in the distributed storage system).

Oisin provides *jobs* as the basic unit of processing in the system. A job consists of a set of *activities* (distributed threads of control) and a set of *contexts* (address spaces, one per node visited by the job). All activities of the job share the context of the job at any common node which they visit. Each job has its own private set of contexts into which the objects it uses are mapped as required. Although contexts may not be shared between jobs, objects may be shared between the contexts of different jobs at the same node.

In this short paper we summarise the experience which we have gained in the design of a native kernel to support object oriented programming. Section 2 outlines the requirements on such a kernel, the main problems encountered and proposes *clusters* as a partial solution. Section 3 describes the use of clusters in Oisin while section 4 describes the relationship between clusters and objects in some detail. Section 5 gives the performance of the resulting system while section 6 provides some conclusions which are motivating our current and future work.

2 The Role of the Kernel in an Object Oriented System.

The COMANDOS project is not intended to provide a single language system, hence one of the main goals of Oisin is to support a range of languages running above and using the basic object management services which it provides. The kernel should not enforce the use of a two level object model at the language level nor, more generally, dictate the granularity of objects. The kernel must support the use of a uniform object model and the use of different language level object models as well as the more traditional operating system functionalities: multiple users, protection and sharing. Sharing is particularly important in Oisin, given the computational model outlined in section 1.

Although there is not yet a large amount of experience with distributed applications running above Oisin, experience with other object oriented systems, notably Smalltalk, suggests that object oriented programs will involve very large numbers of relatively small objects. Moreover many objects will be short-lived and will be known only from a limited scope [5]. It appears obvious, when one considers the overheads that would be involved if the kernel had to manage such objects, that it is inappropriate for the kernel to deal with language level objects. Consider, for example, the sizes of kernel tables that might be required to locate such objects in the distributed system, the problems of virtual memory management for such objects and also the cost of object creation if the kernel had to be informed about every new object.

For these reasons Oisin does not deal directly with objects, although it does know of their existence. The fundamental abstraction managed by Oisin is the *cluster*. A cluster is simply a group of objects which is managed as a unit by Oisin for purposes of location, storage and virtual memory management. Oisin knows that clusters contain named objects (it is responsible for the allocation of global names as required) but does not otherwise know the structure of objects. The object model is implemented on top of the cluster abstraction by the Oisin runtime layer [4]. *A key point is that clusters are transparent to the application programmer who need only deal with objects.*

3 Use of Clusters.

The inclusion of clusters as the basic unit managed by Oisin grew out of the realisation that it was not feasible for an operating system kernel to deal with objects of the expected granularity. However, this is by no means the only motivation for the use of clusters.

Groups of related objects e.g. a closed graph of objects which will be accessed together can be stored in the same cluster [6]. Since clusters are the unit of virtual memory mapping, attempting to map one object of the group will cause the entire cluster to be mapped, thereby minimising the number of object faults.

When mapped, clusters consist of an integral number of virtual memory pages, providing a convenient unit for memory management purposes. Clusters can be implemented in a straightforward manner on top of a segmented virtual memory system such as that provided by Chorus [1].

Many objects are expected to be short-lived and known only from a limited scope. They may never be known outside of the cluster in which they were created and in particular never known to the kernel (for such objects it is also unnecessary to allocate a full global name).

Clusters also provide a convenient scope in which to perform virtual memory garbage collection.

Clusters are the basic unit of sharing between contexts and may be mapped at different virtual addresses in different contexts since they contain (almost) no position dependent information.

Finally, clusters are the basic unit of protection in Oisin. All objects within the cluster have the same protection attributes. This illustrates the basic tradeoff involved in the use of clusters, namely that of granularity versus flexibility. The kernel can be more performant if it can deal with large grained entities but some of the flexibility afforded by the ability to manage objects individually is lost, e.g. it is not possible to assign access rights to an individual object, independent of other objects, unless the object is in a cluster on its own.

4 The Implementation of Clusters.

Clusters are mapped as contiguous regions into virtual memory. All addressing within a cluster is in the form of an offset from the start of the cluster. Thus, as noted previously, a cluster may be mapped at different virtual addresses in different contexts. An exception to this is a special table, (know as the implementation address map), residing in front of a cluster when mapped, which contains the virtual addresses of implementation objects (class or code objects) which have instances residing in the cluster which are bound to the implementation. This table is not shared to allow implementation objects to be mapped at different addresses in different contexts.

A cluster consists of a header and several regions used to store the different parts of its objects.

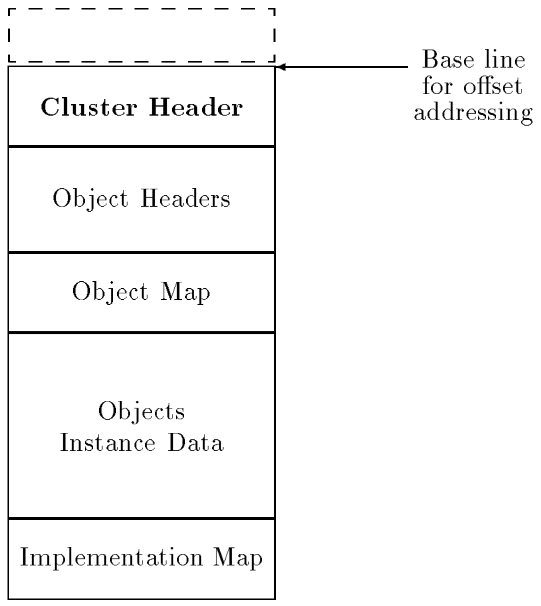

Figure 1: Cluster Structure

Each object in a cluster has a header, giving an offset to the object's instance data as well as other information about the object. All object headers reside within one region of the cluster while the object's instance data resides in another region.

All invocations on an object indirect through the invoked object's header. This allows easy movement of an object's instance data during compaction or on dynamic object growth.

Finally, there is a hash table which is used to locate an object's header given the object's global name, as well as another map, hashed on implementation object name, which is used to bind all instances of an implementation in the cluster to their implementation when an instance is first invoked.

4.1 Object References

An object reference is 8 bytes long. There are two types of object reference. One type is used when refering to an object within the same cluster (intra-cluster reference), the other being used to refer to objects which currently reside in other clusters (inter-cluster reference).

If the reference is intra-cluster then it contains only the offset of the referenced object's header form the start of the cluster.

Inter-cluster references contain the referenced object's global name (allocated by the kernel) as well as a hint for the cluster in which the object resides.

The two types of reference are distinguished by a designated bit.

4.2 The Invocation Mechanism

The runtime maintains a set of registers which identify the object that an activity is currently executing in. These give the virtual addresses of the cluster in which the object resides, the object's header and the object's instance data.

These registers are changed on each invocation and return, as well as by the kernel on object or cluster growth. If the reference used for an invocation is intra-cluster then an inline instruction sequence is used to

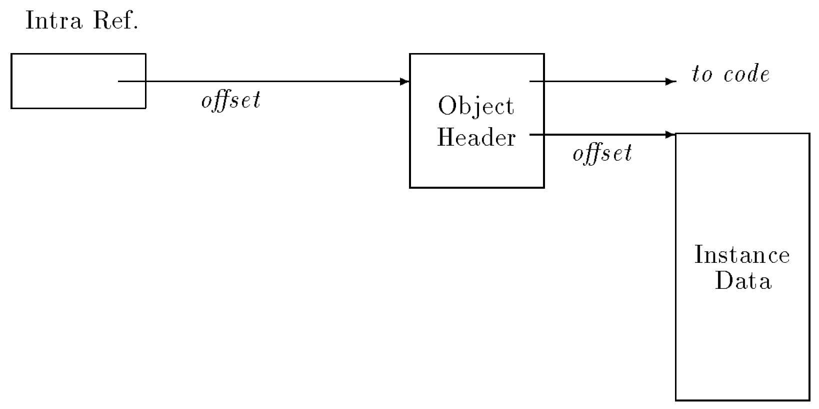

Figure 2: Object Structure

change the latter two object registers and a jump is made into the invoked object's implementation (at the virtual address given in the per cluster implementation address map mentioned previously) where method selection is performed.

If the reference is inter-cluster then a longer invocation path must be taken. The cluster in which the object currently resides is located using the hint contained in the reference which is used to search a per context cluster map.

A cluster fault results if the cluster is not mapped in the context or the object does not in fact reside in the hinted cluster. The kernel is called either to map the cluster containing the object or to perform a remote invocation on the target object.

If the cluster is mapped, the cluster's object map is searched using the global name contained in the reference to obtain the offset of the referenced object's header in the cluster. The activity's current cluster register is updated to reflect the fact that the activity is now executing in a different cluster. The target object is then invoked as in the intra-cluster case.

Inter-cluster invocation returns are trapped to allow location of the invoking object (cluster) which may have been unmapped.

Any references passed as arguments in an inter-cluster invocation are converted to inter-cluster if they were intra-cluster (see below).

4.3 Mature and Immature Objects

When an object is first created it is not visible outside of its initial cluster. The object does not have a global name or any map entries, and may be garbage collected using knowledge available entirely within the object's cluster.

Such an object is said to be *immature*. However, if a reference to an immature object is passed in an inter-cluster invocation then the runtime converts that reference to the inter-cluster form. This results in the referenced object becoming *mature*. It is assigned a global name (by the kernel), given map entries and may now be potentially known from anywhere in the distributed system.

4.4 Clustering Policy and Migration.

Clustering policy is extremely important in maximising the performance of the system. Currently the runtime implements a default policy of creating new (immature) objects in the cluster of the current object at the time of creation. It is also possible for an application programmer to specify the cluster in which new objects are to be created. Application programmers (or even system administrators) can explicitly (re)cluster objects by means of inter-cluster migration primitives which are provided. Finally, automatic grouping of objects into clusters based on static or dynamic analysis of inter-cluster reference patterns is an area for future investigation.

5 System performance.

In this section we briefly outline the current performance of the Oisin runtime:

The performance of the invocation mechanism has proved acceptable. A null intra-cluster invocation was timed at 26 μs on the NS32000 and 31 μs on the MicroVAX II. This can best be compared with a null C function call which costs 12 μs on the NS32000 and 16 μs on the MicroVAX.

Inter cluster invocations cost about 10 times as much as an intra-cluster invocation, largely due to hashing and activity synchronization overheads.

Object creations are hard to time as creating many objects results in the cluster of creation either expanding or being garbage collected. A figure of 950 μs was obtained for the MicroVAX, using a run of 1000 creations, each object being of size 32 bytes, which caused five cluster expansions.

6 Conclusions, Current and Future Work.

Clusters appear to be a useful way of minimising the kernel overheads associated with managing large numbers of relatively small objects. The main disadvantage of the mechanism is the loss of flexibility in managing individual objects.

While performance is acceptable, we are currently investigating the use of virtual addresses in local object references both to improve performance and to allow us to support conventional languages with greater ease. Such support will obviously have major consequences for object mobility within a context and for sharing between contexts.

References

[1] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *Proceeding of the* 12^{th} *ACM Symposium on Operating Systems Principles*, pages 123–136. Association for Computing Machinery, December 1989. ACM Operating Systems Review, 23(5), Special Issue.

[2] Vincent Cahill. OISIN, The Design of a Distributed Object-Oriented Kernel for COMANDOS. Master's thesis, Department of Computer Science, Trinity College Dublin., March 1988.

[3] J. Alves Marques et al. Implementing the COMANDOS Architecture. In *Proceedings of the* 5^{th} *ESPRIT Conference*, pages 1140–1157, Brussels, November 1988.

[4] Andre Kramer. The Design and Implementation of the OISIN Runtime. Master's thesis, Department of Computer Science, Trinity College Dublin., September 1989.

[5] G. Krasner. *Smalltalk-80 : Bits of History, Words of Advice.* Addison-Wesley, 1983.

[6] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2), May 1984.