Experience with Shared Object Support in the Guide System

P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak,
S. Lacourte and X. Rousset de Pina
Bull-IMAG/Systèmes, 2 av. de Vignate, 38610 Gières, France
Internet: {hagimont, krakowiak, rousset}@imag.fr - Phone (+33) 76 63 48 48

Abstract. Support for co-operative distributed applications is an important direction of computer systems research involving developments in operating systems as well as in programming languages and databases. One emerging model for the support of co-operative distributed applications is that of a distributed shared universe organized as a set of objects shared by concurrent activities.

This paper describes our experience in the design, implementation, and use of a distributed system intended to support the above model. The system provides a generic interface designed to support any object oriented language that satisfies a minimal set of assumptions. Shared objects are grouped in clusters; a cluster is implemented as a persistent segment, which may be dynamically mapped in a context (virtual address space) associated with a task. Context dependent information (e.g. protection rights) associated with an object is lazily computed and stored in the context as a separate memory segment.

A prototype version of the system has been implemented on the Mach 3.0 microkernel as a base, and used for simple co-operative applications. Our implementation also demonstrates how an object oriented platform can be supported alongside Unix on a modern micro-kernel.

1. INTRODUCTION

Support for co-operative applications is an important direction of computer systems research, involving developments in operating systems as well as in programming languages and databases. One emerging model for the support of co-operative distributed applications is that of a distributed shared universe organized as a set of passive objects (active agents are define outside of objects) [Bal 92], [Dasgupta 90], [Liskov 92]. In this paper we report on our experience in designing, implementing, and using a system to support such a model. Our goal is to provide an efficient platform for a family of object-oriented languages such as Guide (a language designed by our group [Krakowiak 90]), and a persistent extension of C++. In particular, we wish to enhance sharing and protection, to simplify integration and to improve the performance of complex co-operating applications manipulating a large number of small objects (i. e., about a few hundred bytes on the average). Our target application domain includes office applications, such as a co-operative document editor [Decouchant 93] and a system for document circulation [Cahill 93, chap.

8], composed of groups of interacting data centered tools that are inherently interdependent and have frequent interactions.

This paper describes our experience with Guide-2, the second version of an object-oriented distributed platform intended to support these complex applications. Guide provides a single global address space of potentially persistent objects shared by multithreaded, possibly distributed Tasks[1]. Crucial to the design is the ability to provide mechanisms that allow sharing of objects at different virtual address locations, and efficient management of inter-object reference translations (i.e. swizzling and unswizzling of object identifiers). In addition, these mechanisms should allow dynamic binding (applications written in O-O languages and using persistent objects do not always know statically the type of the objects they manage). Sharing and protection should be defined in terms of objects.

The problems are not fundamentally different from those that Multics [Organick 72] intended to solve 25 years ago, by providing a segmented machine on dedicated hardware. The lessons learned from our experience may be summarized as follows:

- A segmented virtual machine, with a dynamic binding mechanism, provides an adequate support for distributed shared objects.
- Segments may be implemented at a reasonable cost on current hardware (and probably even better on the upcoming generation of machines).
- Shared object structures are a convenient base for the programming of distributed co-operative applications.

The remainder of the paper is organized as follows. Section 2 is a summary of the main design choices and implementation principles of the system. Section 3 describes our experience; it concentrates on three aspects : an evaluation of the addressing mechanisms ; a summary of performance figures ; an assessment of the adequacy of the system for the support of co-operative applications. Section 4 presents conclusions and perspectives.

2. SUMMARY OF THE GUIDE DESIGN AND IMPLEMENTATION

2.1. Main design choices

This section summarizes the main design decisions of Guide. A complete description and justification is given in [Chevalier 93].

Object and execution models

The object model provided by the Guide virtual machine defines basic abstractions for building complex structures. The virtual machine is intended to be used by the run-time system of object-oriented languages (in practice: Guide and an extended C++). The model defines three basic abstractions: *instance-objects*, *class-objects*, and *code-libraries*. The

---

[1]We use Task with a capital T to differentiate Guide Tasks from the Mach tasks used in the implementation.

corresponding entities are potentially persistent; they are named by universal system references. Figure 1 shows the organization of these entities.

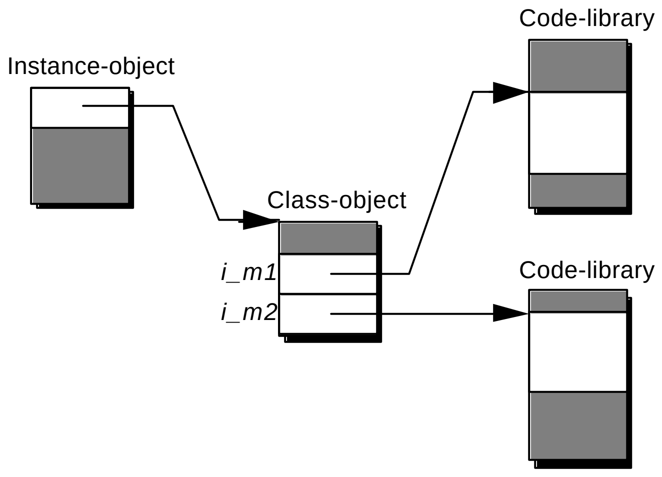


Figure 1: *The generic object model*

Class-objects and instance-objects are defined separately, in order to enforce modularity; the system knows about the link between an instance-object and its class-object. An instance-object can only be accessed using the methods defined in its class. The system does not manage relationships between class-objects. The code of the methods involved in class definitions is stored in code-libraries.

Objects are named by unique system references, and may contain references to other objects. A code-library may contain a reference to a procedure in another code-library. Objects are passive (active agents are defined independently from objects).

The execution model is based on multithreaded Tasks. A *Task* is a set of resources, in particular a distributed virtual address space, shared by its *activities* (sequential threads of control). The address space of a Task is composed of a set of contexts. A *context* is a virtual memory local to a node. A Task may span many nodes and the set of objects it contains can evolve dynamically. In practice, a program is represented by a Task, and a complex application may involve several cooperating Tasks.

Shared objects is the only means of communication between threads within the same Task or in different Tasks. The system should provide different policies to implement object sharing (i.e., one copy for read/write object-instances, multiple copies for class-object)

Management of shared objects

In order to be accessible, an object must be mapped in a context of a Guide Task. Object sharing between Tasks could be implemented either by sharing contexts between Tasks or by mapping an object in separate contexts, one per Task. The second solution was adopted in order to provide protection for individual objects. Tasks do not share contexts and protection is enforced by isolation of Tasks. Furthermore, the protection scheme described in [Hagimont 92] does not allow objects of different owners to be mapped in the same Task context.

Our experience shows that most Guide data objects are small (i.e. less that 300 bytes). Using objects as units of sharing would mean supporting the cost of a mapping for each object binding. We therefore decided to use an object clustering scheme. A *cluster* is a set of (logically related) objects that have the same owner; clusters are the units of mapping. A cluster is mapped in the context of a Task when two conditions are fulfilled: an object of the cluster has been called (for the first time) by an object mapped in the context; the object caller has the same owner as the called object. In practice, clusters can be used at the application level to group logically related objects; the cost of cluster mapping is amortized if most references are local to the cluster.

Object binding

The main motivations in the design of our generic virtual machine [Freyssinet 91] are to provide dynamic binding of references (in order to accommodate polymorphism rules of languages), and to support persistent shared objects that may be used to build more complex structures by embedding references to external objects within the instance data of an object.

This design is based on the following decisions:

- In a previous prototype [Balter 91], each method call was interpreted, i.e. the binding of code and data was checked by the kernel before the actual call. In order to improve performance, interpretation is now only done at first call.
- Since we only have a 32 bit address-space, we reuse space by dynamically mapping clusters in address spaces. An object may be mapped at different addresses, which excludes traditional pointer swizzling. The solution was to simulate a Multics-like segmentation mechanism [Organick 72].

A reference in an object *O1* to another object *O2* mapped in the same address space *A* is made through a linkage segment associated to *O1* in this address space. This linkage segment is built at the first use of *O1* in *A*, using a model generated by the compiler. For each external reference in *O1*, the compiler includes an entry in its linkage segment; this entry is filled (i.e. the reference is bound in *O1*) at the first method call from *O1* to the object pointed to by this reference. After binding, further method calls to the object use indirect addressing through the linkage segment of *O1*, without further interpretation.

In fact, all the abstractions of the virtual machine are managed in this way. A code-library refers to other code-libraries through its linkage segment, and a class-object refers to code libraries in the same way.

2.2. Implementation

Overall architecture

Figure 2 shows a global view of the system.

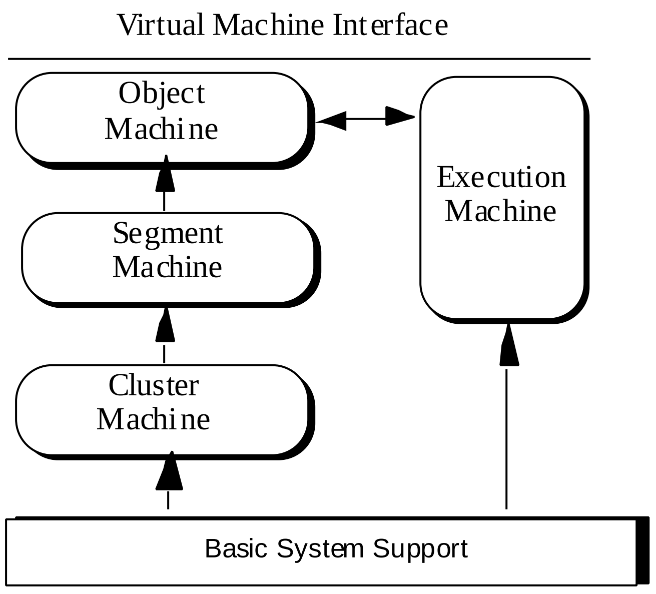


Figure 2: *The global architecture of Guide-2*

The virtual machine provides the interface used by the compilers. This interface gives access to the services that relate to execution management (Tasks and threads management) and object management (object creation and invocation).

The **Object Machine** provides object management facilities. Its basic abstractions are instances, classes and code-libraries. This machine provides primitives that allow the creation of classes and code libraries (for the compilers), and primitives used at execution time such as object creation and invocation.

The object machine is built on top of the **Segment Machine**, which provides the basic mechanisms for sharing and dynamic binding. This machine allows the mapping of segments at different addresses in different contexts, and the dynamic binding of object names at execution time.

The segment machine is built on top of the **Cluster Machine**. The cluster is a group of segments, it is the unit of sharing. This machine provides the cluster sharing mechanism and the management of clusters in permanent storage.

Finally, the execution structures of the system are managed by the **Execution Machine**. This machine implements multi-context Tasks and activities that run in these contexts. It also implements the diffusion mechanism which allows a thread to cross context boundaries in order to run in another context of the same node or another node.

The cluster machine

The cluster machine is composed of two layers: the lower layer (not described here) is in charge of cluster storage on disk; the upper layer manages cluster sharing between contexts.

Cluster sharing is implemented with the Mach external pager facility. A pager runs on each node and is in charge of the management of a set of clusters. These clusters are mapped in contexts, according to the protection policy.

The cluster machine allows a cluster to be shared between contexts running on different nodes. Thus, the pager which manages this cluster controls the consistency of the shared data.

Globally, Guide provides two mechanisms for cluster sharing: the mapping mechanism and the diffusion mechanism (essentially a remote procedure call in which the server node is determined at run time), which allows activities to share clusters on the same node. The diffusion mechanism is used for clusters that can be modified.

The segment machine

The segment machine provides a segmentation mechanism à la Multics, allowing segments to be shared at different addresses in separate contexts.

The first time a segment *S* is accessed in a context, a linkage segment, local to the context, is created for *S*. An entry in this linkage segment is associated to each external reference to another segment in *S*. All these entries are initialized with a null value.

When an external reference in a segment is used, the corresponding entry in the linkage segment is checked. If its value is null, then dynamic binding of the reference occurs; if not, the entry gives the address of the referenced segment in the context.

Figure 3 illustrates this segmented architecture:

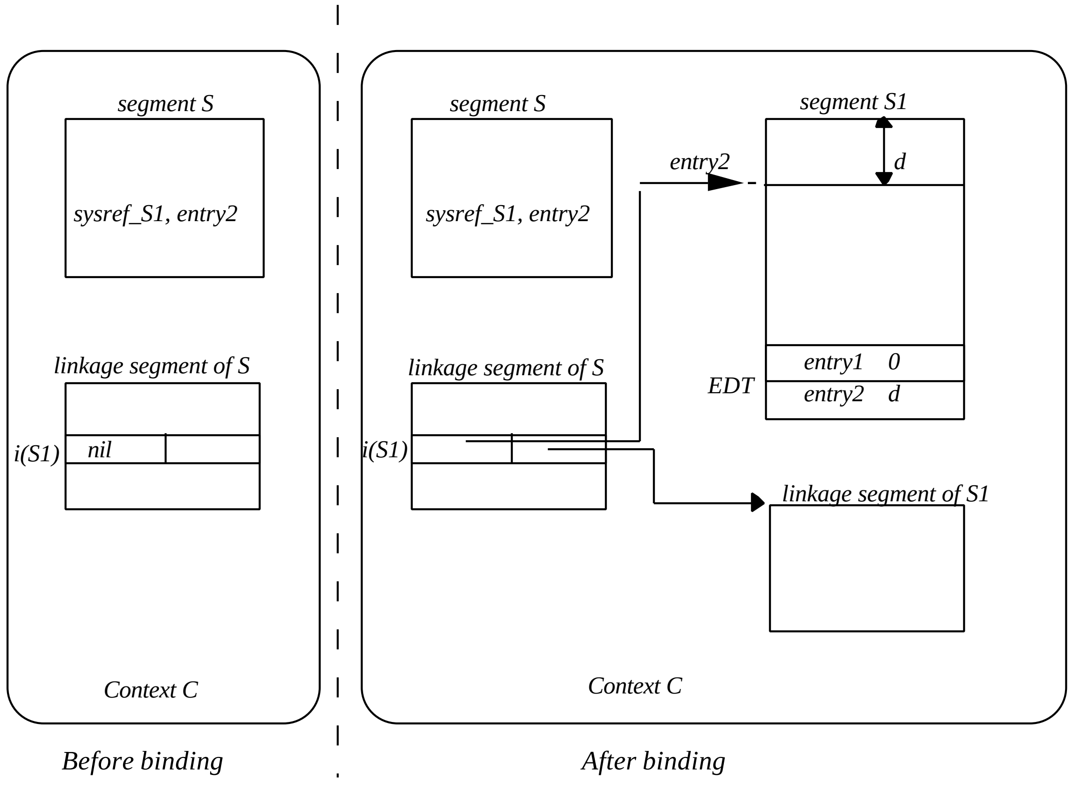


Figure 3: *Binding of an external reference*

The left part of the figure shows the segment *S* before the binding of its external reference to segment *S1*. Note that an external reference is composed of a reference to a segment and the description of an entry point in the referenced segment. Entry *i(S1)* of the linkage segment of *S* is associated with the reference to *entry2* in *S1*.

The right part of the figure shows the context *C* after binding. Segment *S1* (actually, its cluster) has been mapped and a linkage segment has been allocated for *S1*. Entry *i(S1)* in the linkage segment of *S* now points to the exported element *entry2* in *S1*. An entry in a linkage segment has two fields (*s* and *ls*) which respectively record the addresses of the referenced element and of its linkage segment.

In each segment *S*, a table called the *EDT* (Exported Definition Table) gives the offset of the exported element of *S*. The modification of the state of segment *S* only updates its *EDT*. Another, possibly empty, table called the *ERT* (External Reference Table) contains the external references of segments which are likely to be used by *S*; these references will automatically be bound together with *S*.

The linkage segments of the segments bound in a context are dynamically allocated in the virtual memory of the context.

The object and execution machines

The generic object model defines three basic abstractions: instance-objects, class-objects, and code-libraries, as described above. All these entities are implemented as segments.

An *instance* is a segment with one external reference to its class segment. The reference to the class is stored in the ERT of the segment, since an access to an instance always involves an access to its class. An instance segment may contain external references to other instance segments.

A *class* is a segment with one external reference per method, pointing to code-libraries in which the code of the methods is stored.

A *code-library* is a segment that contains the compiled code of some methods. The EDT of a code-segment contains one entry per method. A code segment may contain external references to other code segment.

Figure 4 illustrates the implementation of the object machine on the previously defined segment machine. In this figure, the following references have been bound:

- the reference from the calling object *O* to the called objet *O1*,
- the reference from the called object *O1* to its class *C1*,
- the reference from the called class *C1* to the called method m1.

Thus, if *R* is a register that points to the linkage section of the current object *O*, then the invocation of method *m1* on object *O1* will execute the method at the address:

$R[i(O1)].ls\varnothing ls[i(m1)].s$.

An *object fault* occurs if an unbound reference to an object is used in a call. A *method fault* occurs if a method is called with an unbound reference from its class. Object and method faults are detected at run time, and kernel primitives are called in order to perform the binding.

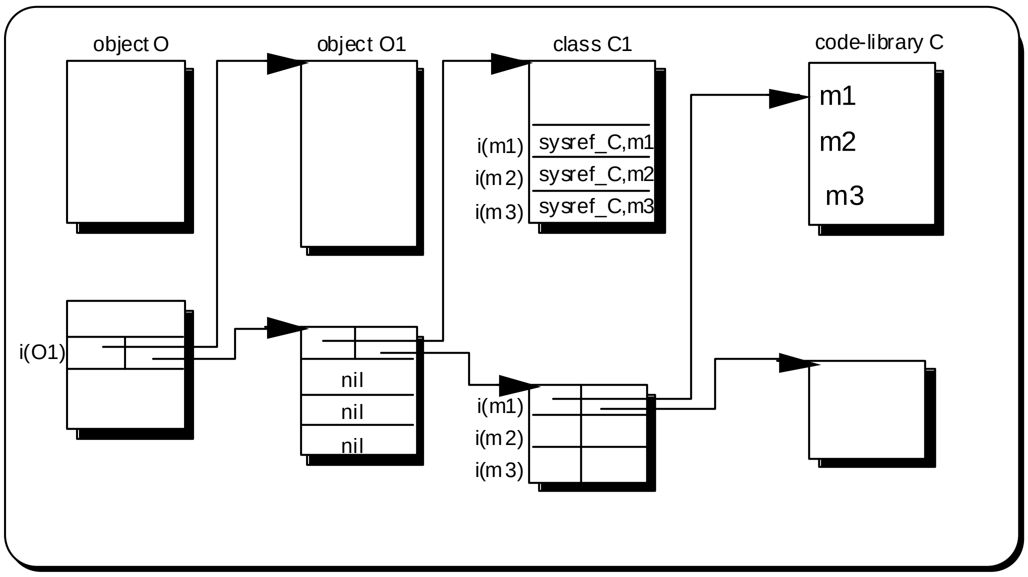


Figure 4: *Segment structures for object support*

After a reference to an object *O1* contained in an object *O* has been reassigned in a context *C,* a context *C'* that shares *O* with *C* may have a linkage segment that still points to *O1* in *C'*. If this reference is used in *C'*, the reference should be explicitly rebound.

The execution machine manages distributed Tasks, contexts and activities. A context is implemented by a Mach task. A Guide Task is implemented by several Mach tasks distributed on the network. Mach threads are used to implement local representatives of activities in contexts. Remote-machine invocations and cross-context invocations are implemented using Mach IPC. The management of Tasks and contexts is performed on each Guide machine by a global daemon.

3. EXPERIENCE AND EVALUATION

In this section, we concentrate on the mechanisms provided by the system for object addressing, and we evaluate the benefits of our design decisions.

In Section 3.1, we develop the object fault and method call mechanisms, with an emphasis on the use of caching. In Section 3.2, we evaluate our design through performance measurements. Section 3.3 is devoted to the presentation of applications developed with the Guide language; this experience provided useful statistics for the validation of the assumptions used in the design of the addressing mechanism.

3.1 Addressing mechanism

The performance of the addressing operation strongly relies on caching. Two caches are managed in each context: a segment cache and a cluster cache. They use locality in order to speed up the processing of segment and method faults.

Object call

In the object virtual machine, three virtual registers are managed:

- **Rbo** which contains the address of the current object. A variable in the state of the object is an offset from this address.
- **Rblo** which contains the address of the linkage segment of the current object. It allows the use of external references stored in the object.
- **Rblc** which contains the address of the linkage segment of the code-segment that is currently executing. It allows the use of external references stored in the code-segment.

Since these registers must be kept for each thread in a context, they are implemented as local variables (on the stack) that are initialized at the beginning of each method.

The parameters which allow to process object faults are grouped in a block (*CallBlock*). A *CallBlock* includes the called object reference, the index of the called method, the entry in the linkage section of the calling object that must be updated (such an entry is called a *Handle*).

When an object fault is detected, the *guide_ObjectFault* () primitive is called to handle the fault. In order to check method faults, the entries of the linkage segment of a class are initialized with a reference to a kernel primitive *guide_MethodFault* ().

The object call described in the previous figure is compiled as:

```
CB.ObjectRef = sysref_O1;             /*  Reference of the called object*/
CB.Method = i(m1);                       /* index of the called method */
CB.ObjectHandle = &(Rblo[i(O1)]);        /* the handle in the linkage section */
CB.Parameters = <parameters>;            /* parameters of  the method call */

if (Rblo[i(O1)].s  !=  <object O1>)  then    /* test the validity of object binding */
        guide_ObjectFault (&CB);
else
        (Rblo[i(O1)].ls∅ls[i(m1)].s) (&CB);  /*  may result in a method fault  */
```

and a method call is compiled as:

```
method M (CB)
{
Rbo = CB-∅ObjHandle∅s;
Rblo = CB∅ObjHandle∅ls;
Rblc = CB∅ObjHandle∅ls∅ls[i(m1)].ls;
< compiled code>
}
```

The *guide_MethodFault* () primitive has the same interface as a compiled method since they are called by the same mechanism.

The *guide_ObjectFault* () and *guide_MethodFault* () primitives both bind a reference respectively to an object and a method, and restart the invocation. These primitives rely on the **reference binding** function provided by the segment machine.

Reference binding

In the segment machine, each reference that has been bound is registered in a cache managed as a hashed table called the *segment cache*. This cache is looked up at each reference binding.

If the lookup succeeds, the cache gives the addresses of the segment and its linkage section in the current context. The handle of the reference that caused the fault is updated with these addresses and the offset of the referenced element in the target segment.

If the lookup fails, the system looks for the mapping addresses of the segment and its linkage segment, and inserts this information in the segment cache. The function which determines the mapping address of a segment is called **segment binding**.

Segment binding

In order to find the mapping address of a segment, the cluster that contains this segment must first be mapped. We do not describe in detail how the system finds the identifier of this cluster (a segment may migrate between clusters). We assume here that the location cluster of a segment can be derived from the identifier of the segment.

We first have to get the mapping address of the cluster that contains the segment, and then to search the segment in the cluster.

The first step, called **cluster binding**, is described in the following subsection.

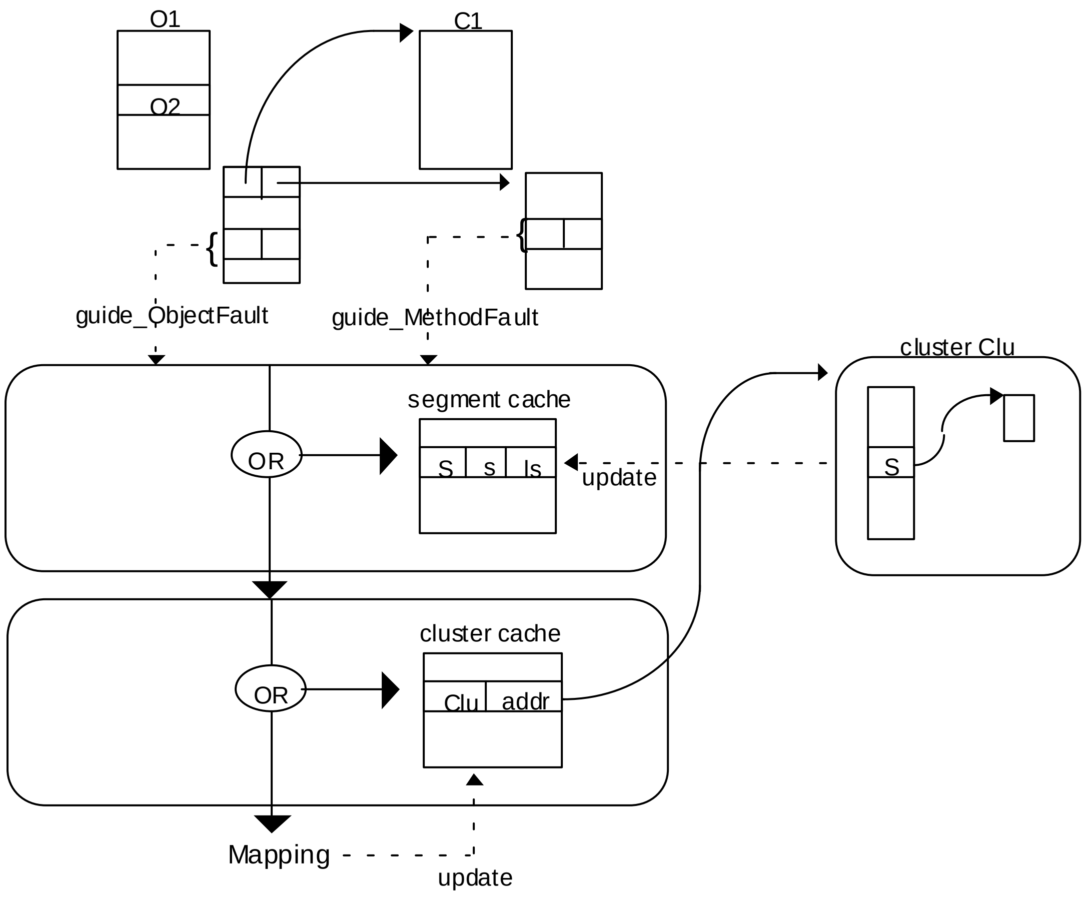
The second step is implemented with another (persistent) hash table managed in the cluster. This table provides, for each segment, its displacement in the cluster.

Cluster binding

The last cache registers the mapping addresses of the mapped clusters in the current context. If a cluster is not found in this cache, then a mapping request is sent to an external pager, in order to ask for the mapping, and the cache is updated.

Global view

The following figure gives an overall view of the addressing process.



3.2 Performance evaluation

The table below gives some performance figures for the steps described above. The machine is a Bull-Zenith P.C. 486 (33 MHz).

Object Call (without fault) (1)	4.4 µs
Object Fault (segment cached) (2)	22 µs
Object Fault (segment not cached, cluster cached) (3)	55 µs
Object Fault (cluster mapping) (4)	10 ms
Method Fault (segment cached) (5)	35 µs

An object call without fault (1) does not call any primitive of the Guide kernel. This can be compared to the cost of a procedure call on the same processor (0. 9 µs) and to the cost of the virtual method call on a C++ object (1. 5 µs) where sharing and persistence are not managed.

The object fault when the segment is cached (2) involves only the segment machine. When the segment is not cached (3), the cluster machine returns the mapping address of the cluster that contains the object, the segment is searched in the cluster, and finally the segment cache is updated.

When the containing cluster is not cached (4), the cluster is mapped in the current context, and the cluster cache is updated. Then, the cost of the segment binding is not visible, since the cost of a mapping operation is much greater.

The line 5 gives the similar cost for a method fault. It greater than the previous results (2) because a method fault includes some protection mechanisms that are outside the scope of this paper.

In the worst case, an object fault and a method fault occur in a method call.

Our motivation in the design of the Guide addressing scheme was to avoid the call of a kernel primitive for each method call. Our strategy is based on the assumption that the set of variables that contain an object reference, used for method invocation, is a small set. This means an important locality, not only of the called objects, but also of the variables used for the invocations on these objects. The next section provides some support for this assumption.

3.3 Supporting a real application

The implementation of Guide-2 started at the end of 1991, using first Mach-3.0 with OSF-1/MK-13 on Bull-Zenith P.C. 486 (33 MHz) connected to a 10 Mb Ethernet. A prototype of the Guide system is currently available on these machines and already supports several applications. We have not yet learned all the lessons we could expect from the project, since only few applications are running on the system. However, we have developed a distributed cooperative spreadsheet running on the Guide system, and we have made some preliminary measurements about the frequency of use of object references.

This application consists in a distributed spreadsheet, in which user defined tables are composed of cells, and where cells may be shared between tables. The application allows to create a link from a table to a cell of another table. Access rights may be associated to a cell, in order to control cells sharing.

A typical use of our spreadsheet application is for a network of stores in a country. A table is associated to each store for registering sales. The prices are stored in a shared table (prices are common) whose cells are used for the accounting of each store.

The statistics about the use of external references in the spreadsheet application are in the table given below.

percent. of Object Call with the same reference	85-97%
percent. of Method Call with the same reference	99 %

The results in this table argue favorably for the hypothesis we made about the locality of the external references used for object invocation. This preliminary result should be tempered by the fact that the application has a small number of classes; measurements on larger applications will be the subject of further experiments.

However, the locality assumption is not always valid for applications that declare many object references on the stack. For instance, we made the same measurements with a recursive version of a simple program, the Hanoi Towers, where many object references are parameters on the stack. Thus, the handles used in the addressing process for these objects are also on the stack, and are often used only once. In this case the statistics are the following:

percent. of Object Call with the same reference	37 %
percent. of Method Call with the same reference	99 %

The locality for object references has decreased, but is still good. We can notice that statistics for method references are still good, regardless of the reference to the called object; the reference to the method code is bound only once in the linkage segment of the class. The linkage segment of a class is shared by all the instances of this class mapped in the same context.

The last application we experimented with is the Cattell benchmark [Cattell 92], used by practitioners of database systems. More precisely, this application implements a travel in a graph in which each node has three children nodes randomly chosen among 1000, 5000 or 8000 nodes, and the travel is done down to seven levels (a total of 3280 nodes are visited). We obtained the following measurements:

percent. of Object Call with the same reference for 1000 nodes	91 %
percent. of Object Call with the same reference for 5000 nodes	71 %
percent. of Object Call with the same reference for 8000 nodes	58%
percent. of Method Call with the same reference	99 %

The probability for two nodes to have a common child decreases when the node number increases.

Globally, the preliminary lessons we learned from these experiments are that the validity of our design hypothesis depends on the application type, but the results are very good when the supported application manages complex graphs of persistent objects, and acceptable for applications that manage simple object that are often exchanged as parameters.

4. CONCLUSIONS

In conclusion, we first summarize the results of the evaluation of the basic addressing mechanisms for a distributed object-oriented system; we next present some elements of our experience in implementing the Guide-2 system on top of the Mach 3.0 micro-kernel; we finally outline our plans and perspectives for the continuation of this work.

4.1. Lessons learned

The basic message of this paper was to present the design choices of a kernel for the support of distributed objects, and their impact on the performance of object management. As a first step in the validation of this design, we developed simple co-operative distributed applications that use object persistence. The main design decisions may be summarized as follows:

- Objects are managed as segments. A linkage segment is associated to each segment, and segment addressing is performed through handles stored in this linkage segment.
- The system detects and handles two kinds of events in the addressing mechanism: object faults and method faults.
- Faults are handled by the Guide kernel, in which two caches are managed in each context: the cluster cache that registers the clusters (a set of interrelated objects, and the unit of mapping) mapped in the context; the segment cache that registers the segments which have already been bound in the context.

The performance measurements show that, after a reference to an object has been bound, the cost of a method call is very low, considering the provided functionalities (sharing and persistence). The first statistics we got from experimental applications running on the Guide system validate the assumptions about the locality of the references used for object invocation.

4.2. Experience with Mach 3.0

A more complete discussion about the adequacy of Mach 3.0 for the support of an object-oriented distributed system can be found in [Balter 93]. We give here a summary of this analysis:

- Since the Guide model can be viewed, as far as the overall execution structure is concerned, as a distributed version of the Mach model, the mapping of the Guide abstractions (Tasks and Activities) on the Mach abstractions (tasks and threads) is straightforward.
- The Mach port abstraction, which provides a location transparent address for message passing between tasks, allowed us to develop and debug the entire Guide kernel on a single machine, since message passing primitives between tasks on a single machine and on different machines have the same interface.
- Mach ports are protected in the sense that a port cannot be used unless it has been explicitly given by a task that has the required rights on this port. This allows the implementation of the protection requirements in our system.
- The ability to design our own memory manager was one of the key benefits from using the micro-kernel approach. It allows a simple implementation of object sharing between different nodes, which was not straightforward on Unix, and the implementation of flexible consistency policies according to application requirements. It also allows the efficient management of fine-grained objects. The use of

memory managers allows a clear separation between the object as unit of addressing, the cluster as unit of mapping, and the page as unit for I/O transfers. This allows an optimization of the multiple facets of memory management.

- The ability to create a task on a remote node would have greatly simplified the overall architecture and especially the management of the execution structures.
- Protected ports have a major drawback in a distributed environment: applications cannot share ports.
- Port group would have been convenient for managing distributed entities or providing fault tolerance facilities.
- As proposed in [McNamee 90], it would be interesting to provide the ability to manage page replacement in physical memory at the level of a memory managers, for the improvement of cluster paging.

4.3. Perspectives

The Guide system is currently used for the development of new experimental applications, and as a platform to which we can add new functionalities in different areas.

The system provides a generic virtual machine for the support of several object-oriented languages. The applications we developed were written with the Guide language. We are currently working on the support of a extension of the C++ language that would manage shared persistent objects. We are also working on the improvement of the Guide language.

Work is also in progress to integrate transactions in the Guide system.

In a further development of the project , we intend to use the upcoming 64 bits architectures for the support of our generic virtual machine architecture. As was shown by several projects [Chase 92][Heiser 93], significant performance gains may be expected.

Availability Papers written in English describing the Guide system and the Guide language are accessible via ftp anonymous on the machine imag.fr. They are stored in the directory:

pub/GUIDE/doc

Acknowledgments. The work described in this paper has been partially supported by the Commission of European Communities through the Comandos ESPRIT project. J. Cayuela, M. Riveill, and M. Santana contributed to the design described in this paper. F. Saunier implemented the co-operative spreadsheet application. In addition we would like to thank J. Mossière and the referees who provided detailed comments on our paper.

REFERENCES

[Bal 92]

H. E. Bal, M. F. Kaashoek and A. S. Tanenbaum, Orca: a Language for Parallel Programming of Distributed Systems, IEEE Transactions on Software Engineering, 18 (3), pp. 93-112, 1992

[Balter 91]

R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de

Pina, R. Scioville, G. Vandôme, Architecture and implementation of Guide, an object-oriented distributed system, *Computing Systems*, vol. 4, 1, winter 91, pp. 31-68

[Balter 93]

R. Balter, P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte and X. Rousset de Pina, Is the Micro-kernel, Technology Well Suited for the Support of Object-Oriented Operating Systems: the Guide experience, accepted at *Workshop on Microkernels and Other Kernel Technologies,* September 1993

[Cahill 93]

The Comandos Distributed Application Platform, V. Cahill, R. Balter, X. Rousset de Pina and N. Harris (Editors), Springer-Verlag [to appear, 1993]

[Cattel 92]

R. G. G. Cattell and J. Skeen, Object Operation Benchmark, *ACM Transactions on Database Systems*, 17 (1), Marsh 1992, pp 1-31

[Chase 92]

J. S. Chase, H. Levy, M. Baker-Harvey, E. D. Lazowska, Opal: A Single Address Space System for 64-bit Architectures, *Proc. of The Third Workshop on Workstation Operating Systems (WWOS III),* Key Biscayne, Apr. 1992, pp 80-85

[Chevalier 93]

P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte and X. Rousset de Pina, Supporting shared persistent objects in a distributed system, Bull-IMAG technical note (submitted for publication)

[Decouchant 93]

D. Decouchant, V. Quint, M. Riveill, I. Vatton, Griffon: A Cooperative, Structured, Distributed Document Editor, Bull-IMAG Technical Report, 93-01, May 93, pp 1- 30

[Dasgupta 92]

P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, U. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, C.J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System, *Computing Systems*, vol. 3, 1, pp. 11-46

[Hagimont 92]

D. Hagimont, S. Krakowiak and X. Rousset de Pina, Protection in an object-oriented distributed virtual machine, *3rd Internat.Workshop on Object Orientation in Operating Systems*, Paris, Sept. 1992, pp 273-277

[Heiser 93]

G. Heiser, K. Elphinstone, S. Russell and G. R. Hellestrand, A Distributed Single Address-Space Operating System Supporting Persistence, *SCS&E University of New South Wales*, Report 9302, March 1993, pp 1-14

[Krakowiak 90]

S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, Design and implementation of an object-oriented, strongly typed language for distributed applications, *Journal of Object-Oriented Programming*, 3,3, Sept.-Oct. 1990, pp. 11-22

[Liskov 92]

B. Liskov, *Preliminary design of the Thor Object-Oriented Database System*, Programming Methodology Group Memo 74, Laboratory of Computer Science, MIT, 92

[McNamee]

D. McNamee and K. Armstrong, Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies, *Proc of the Mach Usenix Workshop*, Burlington, VE, Oct 1990, pp 31-43

[Organick 72]

E. I. Organick, *The Multics system: an examination of its structure*, MIT Press, 1972