Distribution and Persistence in the IK Platform: Overview and Evaluation*

Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques IST-INESC, Portugal

ABSTRACT: IK is an object-oriented platform that simplifies the construction of applications that handle persistent and distributed data. A single programming paradigm is used to manipulate volatile, persistent and distributed objects uniformly. Object invocation is the basic primitive of the system, embodying all the features required for transparent handling of persistence and distribution. Object references can be freely passed as parameters in remote invocations or stored persistently.

Objects are initially created as volatile and are automatically recycled when they are no longer referenced. Objects are promoted to persistent when they become reachable from a persistent root. Persistent objects are read from disk on demand when they are invoked and are written back to disk when the application terminates.

Applications are written in a language similar to C++, but with some semantic differences. Classes are represented as special persistent objects. They are dynami-

*This work was partially supported by CEC under ESPRIT contracts n° 2071, COMANDOS, and n° 5279, HARNESS.

cally linked when needed and can be updated at run-time to install a newer version of the code.

This paper contains a technical description of IK and an evaluation based on two years of experience in using the system to build applications.

1. Introduction

A major source of complexity in distributed applications is their need to handle distributed and persistent information. This task may require three different programming paradigms and languages: one for in-memory computation, another to access remote entities and a third to manipulate persistent information.

Environments such as DCE [Ope91] provide an architecture and a set of tools to build distributed applications based on the client-server model. Entities specify their interfaces using an Interface Description Language and communicate through remote procedure calls. Nevertheless, the client-server model does not provide a complete and uniform programming paradigm, as promised by object-oriented models. Several approaches have been followed to extend the object-oriented model to distributed environments. A line of research has been the direct support of objects by the operating system [Liskov 87, Dasgupta 88]. However, due to the cost of such objects, this approach is only feasible for coarse-grained objects. Other projects chose to define new object-oriented languages to support distributed objects [Black 86].

Persistence is an orthogonal attribute to distribution and has been addressed in a number of ways. Distributed file systems offer a first level of data sharing among several machines. Persistent languages such as PS-Algol [Atkinson 83] provide transparent support for persistence of language level objects.

Few have addressed distribution and persistence of fine-grained objects in a uniform model, where object references can be freely passed in remote invocations and stored persistently. We believe that this single and uniform programming model for the manipulation of volatile, persistent and distributed objects may have a positive impact on the productivity of application development. The same model could be used to program all building blocks of an application from the fine-grained, language level, objects, to coarse-grained servers.

Our main objective in IK is to experiment with a uniform programming paradigm for developing distributed and persistent applications. We want to provide system support for the fine-grained objects manipulated by the programming language. These objects are accessed and manipulated independently of their location, and references to them can be passed freely as parameters in invocations and stored persistently. Objects are persistent if they are reachable from an eternal root. The platform reads the objects from secondary storage on demand when an operation on them is invoked, and stores them back to disk either under programmer's initiative or when the application terminates.

IK is mainly targeted to cooperative applications executing in a local network comprising a few tens of workstations under a common administration policy. Applications are written in EC++, a language similar to C++ with some semantic extensions and restrictions. The programming environment is based on a version of the ET++ [Gamma 88] library ported to our platform, providing a framework for application development. Some of the applications built on top of IK are a cooperative calendar manager that allows several users to schedule and negotiate meetings, a cooperative graphical editor that supports multiple users concurrently, a browser and an inspector for application development and a tool for the analysis, design and implementation of distributed applications using a uniform high-level description and semantic replication.

The platform consists of the EC++ compiler and a run-time library that handles local object creation, invocation and garbage collection. A set of system services handle remote object location, persistent object storage and naming. IK runs on a network of Sun 3, SPARC stations and Bull DPX2 workstations over various versions of Unix. A port to the Mach 3 operating system was recently completed taking advantage of the more advanced facilities for object identification, location and communication [Castro 93].

The work started four years ago within the framework of the Comandos [Marques 88] Esprit project. The overall goals of the project, the object model, architecture and different implementations were already presented in previous papers [Marques 89, Krakowiak 90]. In this paper we review some of the more important decisions taken in IK, provide an overview of the implementation and evaluate the main design decisions. The structure of the paper is as follows: Section 2 describes the major decisions and guidelines in the design and implementation of IK. EC++ is presented in Section 3. Section 4 is a technical description of IK, and in Section 5 we evaluate the platform based on our experience using the system to program distributed applications. Section 6 presents the related work. Finally, we draw our conclusions in Section 7.

2. *Major Decisions in IK*

The Model

In our model, objects are uniquely identified through long lived, context independent references, called Low Level Identifiers (LLI). LLIs are addresses in a global object space that can be used for equality tests and to trigger invocations on objects. Objects are instances of classes with an external interface exclusively composed of methods. There may be several implementations of the same class, called implementation objects.

In IK persistence is orthogonal to the object's class and is a dynamic attribute of objects. IK has an eternal root, and objects are maintained persistently while reachable (either directly or indirectly) from the eternal root. Objects are discarded when no longer reachable from the eternal root. The system provides primitives to insert and remove references into and from the eternal root. Thus, programmers can store object trees persistently just by naming their root object.

Computation is animated by special active objects: Activities and Jobs. Activities are independent threads of control. A job may contain one or many concurrent activities, executing in different nodes.

Object invocation is the basic primitive of the system, embodying all of the features required for transparent handling of persistence and distribution. We tried to make this primitive as general as possible, providing:

Uniform access. All objects are accessed uniformly regardless of their physical location. The platform detects *object faults*. Any access to an object not in local memory raises an object fault. When handling an object fault, the platform may read the object from secondary storage or may forward the invocation to the node where the object is mapped.

Location independence. The programmer does not need to establish the location of a target object explicitly. The invocation mechanism is uniform for local and remote objects, handling distribution, heterogeneity in data representation, and communication protocols.

Transparent sharing of objects. Objects may be shared transparently when invoked. The computational model provides basic synchronization mechanisms upon which more complex mechanisms can be explicitly programmed.

The decision to handle persistence and distribution solely within the invocation primitive is also justified by implementation issues. To allow an object to access directly the instance data of another object regardless of its physical location, IK would need a mechanism to detect accesses on each pointer dereference. Using a software check on each access was considered to be too inefficient. A solution based on virtual memory techniques to generate and catch an exception when the instance data of a remote or persistent object was accessed would be hard to implement in a efficient way given the low granularity of our objects and the lack of specific operating system support. Thus, we chose to detect object faults only through object invocation (see section 4), and impose strict encapsulation in the model.

The Language and the Run-Time Environment

The coupling between the programming language and the underlying support mechanism is a fundamental issue given our global objectives. An integrated approach in which a new language and its support are developed together is a possible solution. This was the path followed by Emerald [Black 86] and, within COMANDOS, by GUIDE [Decouchant 88]. Although this is an interesting line of work, acceptance of a new language is not easy for the user community.

Another approach is the implementation of the model in a language independent run-time support library. A major advantage of such a library is the possibility to support different languages. We decided to implement a Run-Time Support (RTS) library, that client languages should call to manipulate IK objects. We experimented with the Objective-C [Cox 86] and C++ programming languages, but C++ was our final choice because of its wide availability and growing user community. Although we wanted to provide maximum compatibility with standard C++, complete support of its semantics would be too costly. Therefore, we decided to impose some restrictions on the language to achieve acceptable performance.

The Implementation

The first major decision was related with the implementation of the global address space defined in the model. A virtual address space shared by all applications could be used to implement such object space, as in Amber [Chase 84]. However, we decided to keep both spaces apart because the 32 bit architecture of existing machines is clearly insufficient to implement a persistent object space. To improve performance, we identify objects inside an application address space (a *context*) with memory pointers. Pointers are not valid outside contexts and LLIs are not valid pointers. Hence, conversion between the two formats is required whenever a reference crosses context boundaries.

The second major decision was related with the implementation of persistence. Although the model does not distinguish volatile objects from persistent objects, volatile objects are expected to die young while persistent objects are expected to live much longer. We decided to exploit their different characteristics at implementation level; when created, objects are volatile; objects are only promoted to persistent when they become known outside a context. This way, we avoid promoting objects that were temporally reachable from the eternal root during the execution of the application.

We also took the pragmatic decision to use existing technology in the implementation of the platform: thread packages to support activities within a process, distributed file systems to support persistent data, and RPC packages to implement the communication mechanisms. We also decided to rely on the traditional protection mechanisms provided by the underlying operating system to enforce protection in the platform.

3. An Overview of the EC++ Language.

The EC++ language [Sequeira 91] maps IK concepts onto C++ concepts. IK classes and methods are mapped onto C++ classes and virtual member functions, respectively. Implementation objects result from the compilation of EC++ classes. IK objects correspond to C++ instances of classes. Finally, IK object references are mapped onto typed variables in C++, either pointers to classes or references to classes.

When compiled with the EC++ compiler, classes derived, at some level, from the IK top class (`Aida`) will call the IK run-time, while classes not derived from `Aida` will be processed as standard C++. The former benefit from the extended functionality of the system, but must conform to a set of restrictions.

Two major restrictions are imposed:

- **Objects are not lvalues.** Objects must always be accessed through object references, they cannot be handled as values. Therefore, objects cannot be returned as a result of a method invocation or even be the value of automatic variables. In such cases, object references should be used instead of objects.

  This restriction avoids the promotion of objects allocated in the stack (e.g. local variables, temporary objects created by the compiler) to persistent objects. To allow such promotion, the EC++ compiler would have to intercept all the situations where these objects are created, making the compiler overly complex. We decided to keep it simpler and consider that `Aida` objects must be explicitly created using the *new* operator.
- **Strict Encapsulation.** Only an invocation on an object can access the object's instance variables. There are no pointers to member instance variables. To read or write other object's instance variables, the programmer must define specific methods, hereafter known as `get` and `set` methods, and invoke them explicitly.

  Although the IK model prevents an object to access the instance data of another object, we could allow instance data accesses at the language level, by making the EC++ compiler generate and invoke the `get` and `set` methods automatically. However, we did not want to hide a potentially costly operation, eventually a remote invocation, under a simple pointer dereference in application source code.

Applications are organized as a collection of cooperating classes. Different classes are compiled into different files that are registered in the system as being the implementation objects of the classes. Classes are fairly independent modules; adding or removing methods to or from a class only requires the recompilation of that class. Obviously, changing the interface of existing methods may certainly cause run-time errors. Adding or removing instance variables to or from a class also requires the recompilation of derived classes. A single binary (`ik`) is used to launch all applications, and objects and implementation objects are mapped on demand.

To better illustrate the look-and-feel of the EC++ language and the impact of its restrictions, we present a very simple editor in which multiple users can edit a single shared line of text concurrently. The application consists of a class *Line,* which implements a fixed array of characters, and a class *Editor,* which updates and displays a *Line* object. The example uses the Activity and Lock library classes to express concurrency and synchronization. It also uses the Name Service reference (`NS`) to associate user defined strings to object references. An object registered in the Name Service becomes part of the eternal root set.

Class *Line* (Figures 1 and 2) inserts characters and returns the contents of the entire line. A *Lock* object is used to serialize concurrent insertions. The first EC++ restriction forces us to declare and return the *line_t* type rather than a *Line* instance.

Class *Edit* is presented in Figures 3 and 4. The method *createLine* instantiates a new *Line* object and assigns a name to it (which becomes persistent). The *edit* method obtains a reference for the *Line* object given its name, creates an activity to display its contents each second, and then inserts the characters keyed in into the *Line* object. The second EC++ restriction prevents the editor of directly accessing the contents of a Line object, regardless of the declaration scope of its instance variables.

```
#include "Aida.h"
#include "Lock.h"

const int lsize = 80;
typedef struct {
char  c[lsize];
}  line_t ;

class Line: public Aida {
    line_t text;
    int    index;
    Lock  *lock;

 public:
    Line();
    virtual void   insert(char c);
    virtual line_t getLine();
    virtual void   afterMap();
    virtual void   beforeUnmap();
};
```

Figure 1: Line.h

```
#include "Line.h"

Line::Line()
{
    lock = new Lock;
    index = 0;
}

void  Line::insert(char c)
{
    lock->lock();
    if(index<lsize) {
        text.c[index++] = c;
        text.c[index]   = 0;
    };
    lock->unLock();
}

line_t Line::getLine()
{
    return text;
}

void Line::afterMap()    //
{                        // the use of these two
    lock = new Lock;     // methods is explained
}                        // in section 5
                         //
void Line::beforeUnmap() //
{                        //
    lock = 0;            //
}                        //
```

Figure 2: Line.C

```
#include "Aida.h"
#include "Line.h"

class Edit:public Aida {
    Line * line;

 public:
    virtual void   createLine(char * name);
    virtual void   edit(char * name);
    virtual void   display();
};
```

Figure 3: Edit.h

```
#include "stdio.h"
#include "Edit.h"
#include "Activity.h"
#include "NS.h"

extern void sleep(int);

void  Edit::createLine(char *name)
{
    NS->bindName(new Line, name);
}

void  Edit::edit(char *name)
{
    Activity * act = new Activity;
    char c;

    line = (Line *) NS->lookUpName(name);

    // start method display
    act->start(this, "display", (void*)0);

    while((c = getchar()) != EOF)
        line->insert(c);
}

void Edit::display()
{
    while(1) {
        line_t l =  line->getLine();
        printf("%s\n", l.c );
        sleep(1);
    }
}
```

Figure 4: Edit.C

To execute the "`Edit::createLine`" method, users issue the command "`ik -c Edit -m createLine aName`", creating a persistent *Line* instance named "`aName`". Then, several users may run the edit method to access the Line object. The first editor to dereference the line reference (either in `line->insert` or `line->getLine`) will map the object locally. In the other editors running in the same or different (possibly heterogeneous) machines, these invocations will be forwarded to the editor mapping the line object. When the editor mapping the line object exits, the object is saved and remapped in the next editor invoking it.

4 Architecture and Implementation of IK

The architecture is composed of a *Run-Time Support* (RTS) handling objects within an application address space (a context), and an *Object Kernel* implementing the object address space based on LLIs. Within the RTS we can distinguish an invocation mechanism, a garbage collector of volatile objects, a dynamic linker and an object clustering mechanism. The Object Kernel is internally composed of four components: the Activity Manager is in charge of the active entities; the Storage System (SS) provides support for persistent objects; the Name Service implements a single hierarchical name structure, similar to that of the UNIX file system, spanning machine boundaries; and the Communication Service offers a generic RPC interface independent of the underlying protocol stacks. All these components cooperate to support object identification, localization, naming, storage and remote invocation.

Upon creation, objects are only known within the context of the creating application, and are referenced by memory pointers. We call these objects volatile because they can be recycled by the local garbage collector. If a reference to a volatile object crosses context boundaries, an LLI is assigned to the object and sent instead. The object becomes known to the Object Kernel and reachable from any other context in the network. We call these persistent objects, because they are written back to the SS at application termination, together with the ones they refer.

4.1 The Run Time Support

Object Structure

Objects are represented in a context by a *Run-Time Header* (RTH), to which local references point to (see Figure 5). If the object is mapped in the current context the RTH data field points to the instance data of the object, and is zero if the object is not mapped. This indirection increases the cost of object data accesses, but simplifies object mobility. Likewise, the cache field points to the class method cache if the class is mapped or, otherwise, to an always invalid cache.

Objects support a default set of upcalls, allowing both default RTS management and class specific management when required. The upcalls are divided into two groups: those that allow the RTS to scan references and convert objects between heterogeneous data representations; and those that allow the user to define specific actions to be taken when objects are created, mapped, unmapped and deleted. A default implementation of these methods is automatically generated by the EC++ compiler. The programmer may have to overwrite them to tackle specific cases.

Object References

Object references are direct pointers to RTHs and are used in applications to invoke and refer to objects. As the Object Kernel identifies objects through LLIs, the RTS must convert LLIs into local references before the object is accessed. A common technique [Atkinson 83, Kaehler 86] is to convert global identifiers to virtual addresses when objects are brought into memory. Recently, this has become known as pointer swizzling.

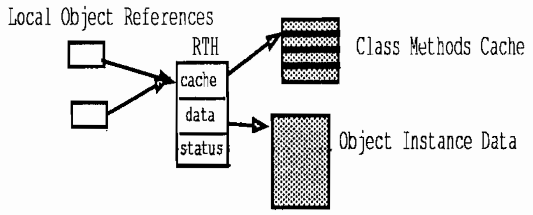

Figure 5: In Memory Object Structure

Pointer swizzling can be implemented in two basic ways. Global identifiers are converted to pointers just before being used by the application (*on demand*) or immediately after being mapped (*on mapping*). Conversion on demand has been used in several systems [Atkinson 83, Richardson 89, Black 86], where a software check precedes each pointer dereference. If the pointer is a global identifier, the object, or a surrogate, is mapped and the global identifier is replaced with the local pointer. In contrast, conversion on mapping ensures that only local pointers are seen by applications, avoiding the checks on pointer dereference. A major disadvantage of conversion on mapping is the need to follow pointers extensively during pointer swizzling. This forces the complete mapping of object trees, leading to immediate and unnecessary conversion of references. Such mapping and conversion wave can be bounded to the size of pages [Wilson 91], which may still lead to a significant amount of unnecessary conversions, because pages are much larger than the fine-grained objects we support.

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

Our RTHs are untyped surrogates that can be generated and reused regardless of the object type they represent. This makes pointer swizzling a much faster operation than in systems that use proxies or typed surrogates [Shapiro 89, Wilson 91, Cahill 90]. In fact, swizzling from global references to pointers to typed proxies may be a very expensive operation. It requires determining the type or class of the global reference, mapping the corresponding proxy class if necessary, and creating a new proxy for the object. All this work is wasted if the object is not invoked.

Method Invocation

Object invocation is performed by searching the method in the class hierarchy and jumping to its address, using a mechanism similar to Objective-C [Cox 86]. Methods are identified by string names. To speed up method lookup at run time, strings are converted to 32 bit identifiers when implementation objects are loaded. Method identifiers are class independent values, unique within a context.

As recently invoked methods are more likely to be invoked again, a standard technique to improve performance is to cache them in a *method cache*. Traditional implementations of method caching use a global cache, where methods of different class hierarchies are stored together [Goldberg 83, Cox 86]. To check if a given method is the right one, the cache uses a key based on both the class and the method identifier. We decided to use a cache per class, rather than a global cache. The key to access the cache and the cache hit check only uses the method identifier, rather than both the method and class identifiers.

We use cache misses to detect object faults with almost no cost at all. As mentioned before, RTHs of objects that are not mapped in the current context have a nil `data` field and a `cache` field which refers to an empty cache. When such objects are invoked, there will be a cache miss and only then the `data` field is tested to distinguish between a normal cache miss (not nil) and an object fault (nil). The object fault detection overhead is reduced to a single test per cache miss.

Local Garbage Collector

We implemented a generation scavenging algorithm [Ungar 84], taking into account the multi-threaded and persistent environment [Ferreira 91].

For the purpose of local garbage collection, the roots of the object graph consist of persistent objects and references found in the stacks of the various threads, including the processor registers. The stacks are scanned in a conservative way, as in [Weiser 89], where 32 bits values that match the address of object RTHs are considered to be local references.

In order to reduce pauses seen by applications, the garbage collector does not recycle the whole heap at a time. The heap is divided into several regions, each corresponding to a given generation. The garbage collection starts by recycling objects in the youngest region, and keeps going to older regions until enough memory is recycled. To allow younger regions to be recycled without having to scan older regions, references from older objects to newer ones are remembered. In assignments to references stored in the object instance data, the EC++ compiler generates code to test a flag in the object header.

Dynamic Linking

Implementation objects are represented at run-time by extended RTHs (see Figure 6), implementing a Smalltalk like `super` and `meta` class hierarchy.

The RTS maintains a context wide symbol dictionary used to resolve undefined symbols during dynamic linking of implementation objects. On the first attempt to lookup a method of a class, the name of the class method table is searched in the symbol dictionary and, if not found, the corresponding implementation object is mapped and linked dynamically with the running application.

Implementation objects can be removed from an address space, both freeing local resources and allowing newer versions to be mapped. To unmap it, we mark the implementation object RTH and prevent further accesses to its methods, which is achieved by invalidating the class method cache and also all the caches of the classes derived from the updated class.

The new version of the implementation object will be mapped when the next invocation to one of its instances occurs. The old version is deleted when none of its methods are in execution. This is periodically checked by the garbage collector, scanning the stacks of the existing activities and looking for invocation frames with a return address that fits within the limits of the implementation object.

Symbol collision is far more troublesome in systems with dynamic linking, because instead of an error during the development of an

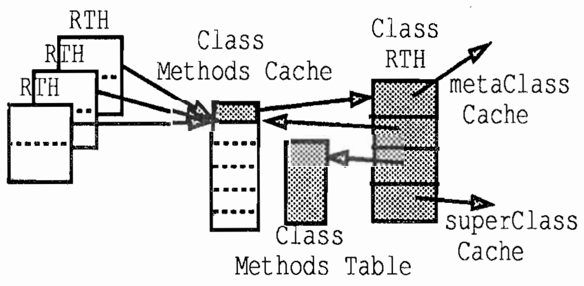

Figure 6: Extended RTH

application, one may encounter unexpected errors at run-time. By making all symbols defined in one implementation object invisible to other objects, with the exception of the method table, we avoid the problems caused by symbol collision, and greatly speed up linking and unlinking.

To support existing libraries, the traditional scopes of symbols is respected when archive files are loaded. However, libraries cannot be updated or unmapped during application execution.

Object Clustering

The goals of object clustering are to reduce the number of objects handled by the Object Kernel, and increase their granularity. If a persistent object is about to be saved, and refers to a private sub-graph of volatile objects, the RTS appends that graph to the persistent object, and creates a cluster. The references in the cluster are swizzled, and the whole cluster is converted to the XDR [Sun86] machine independent data format. Finally, the cluster is presented to the Object Kernel as a single entity, identified with the LLI of the persistent object.

Within a cluster, objects are ordered based on a depth first analysis of the object graph. Volatile objects reachable from more than one persistent object are promoted to persistent, and become roots of new clusters. References between objects in the same cluster are simple offsets within the cluster. References to objects in other clusters are indexes into an LLI table, kept at the beginning of the cluster.

We map a cluster when its root object is invoked. To map a cluster, we allocate space for the whole cluster, using memory mapped files, convert the instance data of the root object to the format of the current machine, and swizzle its references. The other objects in the clusters remain in the external data format and unswizzled until they are invoked. This way we delay touching cluster pages until it is really necessary (Figure 7).

Clusters can be unmapped automatically at application termination, or flushed explicitly during the execution of the application. To unmap a cluster, as the original graph of objects may change during the execution of the application, we may choose either to recreate the whole cluster or simply to append new objects at the end of the original cluster.

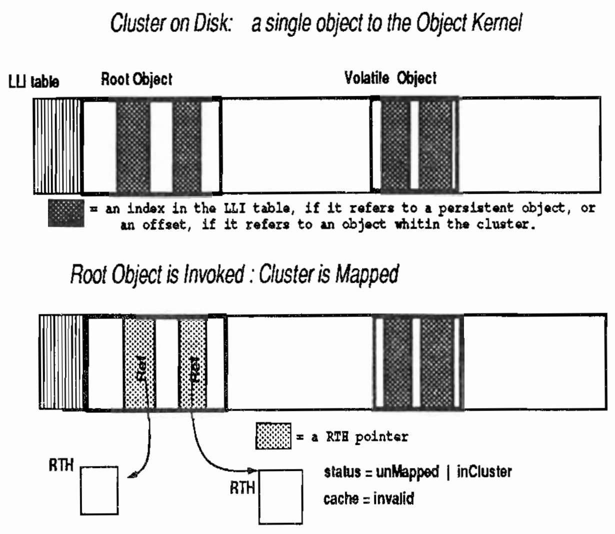
Cluster on Disk: a single object to the Object Kernel
LLI table
Root Object
Volatile Object
= an index in the LLI table, if it refers to a persistent object, or an offset, if it refers to an object whitin the cluster.
Root Object is Invoked : Cluster is Mapped
= a RTH pointer
RTH
RTH
status = unMapped | inCluster
cache = invalid

Figure 7: Mapping a Cluster

Recreating a cluster eliminates objects no longer reachable from the cluster root, keeping clusters compact and increasing locality of reference. Appending new objects to an existing cluster only requires traversing the objects that are mapped, and therefore avoids loading into memory the parts of the cluster that were not mapped. Currently, we have both mechanisms but the decision is hard-wired in the code. We are experimenting a decision mechanism based on the proportion of objects that are still in the XDR data format when the cluster is saved. Objects in XDR format are those that were not invoked while the cluster was mapped. Therefore, if most of the objects in the cluster were not invoked yet, we extend the cluster with new objects, and scanning the object graph stops whenever it reaches an object yet to be invoked. New objects are appended to the cluster, and old ones are saved in their original positions. On the other hand, if most of the objects were already invoked, we simply scan the whole graph, as if we were creating a new cluster.

To maintain the application coherent, whenever an object is saved all objects reachable from it must also be saved, even if they are mapped in remote machines. To propagate the unmapping through the whole graph of objects, IK looks into the LLI table of the original cluster and saves, if necessary, referenced clusters. The same process is repeated in each of the referenced clusters. If the referenced cluster is unmapped, its LLI table is brought into memory and graph traversal continues until the complete graph is saved.

4.2 The Object Kernel

Object Identification.

As mentioned before, within the Object Kernel clusters are single objects identified by LLIs. The hierarchical structure of LLIs simplifies its generation and object location algorithms (see Table 1).

SS Identifier	*SSid*	16 bits
SS Generation Number	*SSgn*	32 bits
RTS Generation Number	*RTSgn*	16 bits

Table 1: Format of LLIs

To allow independent generation of LLIs, the RTS asks for a unique tag (the *SSid* and *SSgn*), and generates LLIs by incrementing the remaining bits of the LLI (*RTSgn*). Once exhausted, another tag is requested. An SS generates tags by appending the *SSgn* counter to its *SSid*.

Object Storage.

Objects are stored as plain files in special directories, assigned to the SSs. The names of the files are their LLIs in a textual (ascii) format, and the directory names encode the SS identifier. We rely on the underlying file system for the actual distribution and protection of persistent data. Clusters are normally mapped in the invoking context. However, if the cluster is already mapped in another context, the SS returns a method stub that is dynamically linked to the invoker, and invocation can be forwarded to the appropriate context.

Object Location.

In the absence of cached information, the location algorithm simply asks the SS indicated by the LLI about the object's current position. The relationship between server identifiers and their communication addresses is kept in files.

If the SS has no record of the object, it assumes the object is still mapped in the creator's context, as indicated by the *SSgn* field of the LLI. The SS maintains the mapping of *SSgn* numbers to context addresses. If the context no longer exists (e.g. it has crashed), the SS assumes the object is lost, and propagates the fault to the application.

To minimize communication with the SS, the RTS holds the object's last known position as a location hint. These hints are passed along with the LLIs during remote invocations. In the current implementation, if different hints exist for the same object, we merely choose the last one received. We may improve hint quality by associating hints with counters as in [Fowler 85]. In the current version of IK, the SS must be informed when an object moves from one context to another. We are improving the location algorithm to allow migration of distributed objects without involving the SS [Black 89].

Object Naming.

The Name Service associates human readable names with LLIs, allowing objects to be accessed given their name. The implementation relies on the underlying file system. To associate a name with an object, the Name Service creates a symbolic link with that name to a file whose name is the LLI of the object. To retrieve the object given its name, the Name Service translates the symbolic link to obtain the LLI of the object.

The use of *links* to hold the (LLI, name) pairs, rather than a small database, allows users to see named objects as normal files. These names may be stored under the user's home directory or in his working area and are searched according to the standard rules of Unix shells. An path environment variable allows users to parametrize the order of these lookups. This approach also makes the Name Service stateless, both simplifying its implementation and increasing its robustness. Note however that objects cannot be accessed from their names using standard Unix commands, because the symbolic links created by the Name Service do not name existing files (they name an LLI). The object referred to by that LLI can only be accessed through the SS interface.

5. *Evaluation of the Platform*

The evaluation of the IK platform is hard due to the multiple issues addressed in its design and implementation. Performance of the basic primitives is important since they support the computation in the model. Another issue is the evaluation of EC++ as a programming language to express computation in a distributed and persistent environment. Finally, since we built the IK platform to assess whether the uniform programming model is effective, we discuss some of our experience regarding the IK programming model.

The evaluation is based on our current set of applications, which are basically interactive applications. They use extensively the version of the ET++ [Gamma 88] library we ported to the platform. The applications and the ET++ library all together comprise 452 classes and 6216 methods, a number we consider meaningful to draw some conclusions. As an attempt to characterize the applications, the calls to RTS primitives were traced during the execution of the applications. The result is presented in Table 2. Due to the transparency provided at the language level, the primitives numerically representative are object creation, object invocation and accesses to objects instance variables. In average, approximately 2% of the primitives called were for object

Relative Usage of RTS primitives	
Object creation	2%
Object invocation	46%
Instance data accesses	52%
Objects	
Class hierarchy level	4.2
Object Size	42 bytes

Table 2: Dynamic Metrics of Applications based on ET++

creation; 46% for object invocation and 52% for accessing object instance variables. We also measured that the average size of objects is 42 bytes and, in average, objects are instances of a 4.2 deep class hierarchy.

5.1 Performance Evaluation

This section presents the performance figures of basic RTS main primitives, implemented on SunOS 4.1.3. The measurements were made using a SPARC station 10/30 with 32 Mbytes of memory for local primitives. In the tests requiring two machines we also used a SPARC station 10/20 with 32 Mbytes of memory, connected by a 10 Mbit ethernet.

Object Creation and Invocation

In Table 3 we compare the cost of the primitives for creating and invoking objects in EC++ and C++. The first test evaluates the cost of object creation and deletion. In EC++, objects were simply created in a loop, and deletion was done by the garbage collector. In C++ the loop also contained an explicit delete to simulate the garbage collector. The figures include the cost of calling empty constructors. In EC++, the creation cost increases with object size, because it increases the garbage collector runs, and with the number of contained references, since references are initialized at object creation time.

Function	EC++ (μs)	C++ (g++)(μs)
Object creation and deletion		
instance data: 2 ints	13	14
instance data: 48 ints + 16 refs	70	16
Empty method invocation:	.48	.34
Instance Data Accesses (integer read) through *this*	.08	.03
Overcoming Strict Encapsulation		
integer read through invocation of `get` methods	.50	
integer write through invocation of `set` methods	.53	

Table 3: Performance of local object manipulation.

Object invocation is often used as a criterion to assess the quality of an object oriented language or system implementation. Empty EC++ virtual method invocation is 41% slower than C++. The EC++ invocation figures represent the optimal case, where the method is found in the cache. Currently, with a 32 entries cache we achieve a 94% cache hit rate. The impact of the remaining 6% cache misses in the overall invocation cost is largely dependent on the number of methods per class and on the number of super classes traversed before the method is found. We measured an average of 8% increase relative to the optimal case.

The next item show the cost to the cost of an object writing an integer value on its instance data through the *this* pointer. The extra cost accounts for the indirection through the object's RTH. Whether the extra cost of IK object structure and primitives is significant in application in-memory computation, it mostly depends on the application characteristics. Furthermore, the exact figure for a given application is hard to measure because instance data accesses are expanded in line in the application code. Nevertheless, in our applications we measured an average of 1050 objects created per second of application execution. Based on the application metrics (table 2), we compute a rate of 24125 invocations and 27300 instance variable accesses per second. By multiplying the operation rates with the corresponding costs, we estimate an overall performance penalty of less than 9%[1]. Such overhead is an acceptable price to pay for a language independent run-time, given the flexibility it provides.

One may still argue the main performance loss results from the strict encapsulation as it increases the number of invocations and the corresponding overheads, such as parameter passing or return values. The last figures presented in table 3 show the cost of reading and writing an attribute of another object through the invocation of its `get` and `set` methods. The C++ figures for invocation of `get` and `set` methods are not presented since C++ does not impose strict encapsulation. The application traces also showed that only 9% of all invocations were intended to read or write other object's instance data, through

[1] $9\% \simeq \frac{(1050*70\ \mu s + 24125*0.51\ \mu s + 27300*0.08\ \mu s)*100}{10^6\ \mu s}$ where 70 μs is the cost to create the largest object we have, 0.51 μs is the average invocation cost including cache misses.

`get` and `set` methods. We think that strict encapsulation is not a significative source of performance loss in our applications.

Garbage Collection

The cost of garbage collection depends on object size, object death rate and on the depth of the object tree. The cost of object creation presented in Table 3 included a minimum garbage collection overhead, because objects did not reference each other, causing a 100% death rate. In practice, the cost of garbage collection is higher; death rates are smaller and object graphs are more complex. An extreme case happens during the creation of large object graphs, where death rates become almost null. In such cases, the garbage collector will be called without memory to reclaim, but will waste time copying created objects to older spaces.

To achieve acceptable performance, the garbage collector must be carefully configured. The trade off is between a low memory usage, and corresponding short application pauses, and better overall performance but with longer pauses. An acceptable configuration for applications based on the ET++ library uses 250Kb for creation space, which typically produces 150 ms pauses every 6 seconds.

Remote Invocation

The cost of an empty EC++ invocation between a SPARCstation 10/20 and a SPARCstation 10/30 is 4.1 ms, almost twice the time of a plain Sun RPC. Table 4 shows that the overhead in remote invocation is for: passing a 128 byte header with context dependent information (0.3 ms); packing and unpacking the header in a universal format, based on method templates (0.2 ms); and finally, delivering `SIGIO` sig-

Empty Sun RPC		2.4 ms
Remote Invocation Overhead:		1.7 ms
Passing a 128 byte Header	0.3 ms	
Header packing	0.2 ms	
Signal delivering and thread switches	1.2 ms	
Total		4.1 ms

Table 4: Invocation between two Remote Objects

nals used in our implementation of the non-blocking redefinition of system calls and the LWP thread switching (1.2 ms).

Thread switching is responsible for most of the overhead. The use of real threads reduces significantly this figure, as happened in the implementation of IK in Mach [Castro 93].

Although the performance of our remote invocation primitive is far from optimal, we could not improve it much further without changing the RPC and thread packages. This would make IK much more dependent on the operating system and machine architecture, which was not the path we wanted to follow.

5.2 Programming in EC++

Several C++ programmers have been using the platform to build applications from scratch. They found the EC++ restrictions forced a different style of programming in C++. Nevertheless, they reported the restrictions do not harm the expressiveness of the language. Programmers found different ways to write their programs, not necessarily more verbose.

More experienced C++ programmers found the style enforced by the restrictions familiar since most of them already used a similar approach. EC++ strict object encapsulation is closer to pure object oriented principles than the C++ class encapsulation, which might be regarded as a compromise between abstraction and efficiency. Since programmers only had to concentrate on higher level decisions and algorithms, leaving the low level details to the IK platform, they reported that EC++ language really helped in the development of the applications.

As mentioned in section 3, EC++ compiles both C++ and IK classes, allowing invocations between their instances. Thus, classes whose instances are known to be always local can be freed from EC++ restrictions, but the programmer must have in mind that they are limited to a single address space. The existence of two semantically different pointers and objects contradicts, to some extent, our goal of providing a uniform and coherent programming model. The decision to allow C++ classes in the platform was pragmatic, justified by the poor performance of early IK releases and by strong restrictions derived from the first EC++ implementation. With the improvements introduced in current IK and EC++ releases, the support of pure C++ classes is now justified for reusability and performance of critical sections.

Extending existing applications designed to run sequentially within a address space, with persistence and distribution, is a different issue. One must study the application and find out the objects that must persist between application runs. These objects should only have instance variables that remain meaningful outside the application context. The accesses to persistent object graphs must be synchronized to prevent against concurrent computation. Finally, one must eliminate the constructs not allowed in EC++ from the application source code.

When early prototypes of IK were released, two Smalltalk programmers initiated the port of the ET++ 1.0 library and corresponding applications to our platform. It took them 5 months to port the 280 classes of ET++ library. In Table 5, we present an effort breakdown of the work carried out. Most of the items are the characteristics of the IK model that implied changes in the source code. The *Other* item accounts for the problems found during the parallel development of the platform and the library.

Item	Changes	Effort
Objects are not lvalues	replace by references	13%
No instance data addresses	replace direct accesses by method invocation	32%
Garbage collection	remove `delete`	1%
Persistence	remove flush and storage code	7%
Concurrency	insert locks	1%
Dynamic Linking	no explicit class code loading, no global variables	0% 18%
Others	bug fixes in IK and ET++	28%

Table 5: Breakdown of the ET++ port.

Concerning the restrictions imposed by our architecture, prohibiting direct access to object instance variables is the most significant. In most cases we simply replaced accesses to object instance variables by invocations to `get` and `set` methods. Only a few, performance critical, methods had to be rewritten.

The second major restriction is related to dynamic linking. The system does not allow direct access to global variables defined in other classes, because they may not be mapped when the variable is accessed, and class mapping is only triggered by method invocation. Nevertheless, despite of the initial cost to encapsulate global variables in a single class, it proved to be helpful in both application organization and debugging.

Most of the ET++ classes are context dependent and their instances are never stored persistently nor referenced from other contexts. In fact, only the 15 classes acting as data repository had to be ported to get a persistent and distributed ET++ library. The remaining of ET++ was ported only to fully test the platform. Since the ET++ experience, other applications were ported with significantly less effort, either by using ET++ or by encapsulating existing C++ classes into a few EC++ classes.

Debugging is a major component in a programming environment. Initially, to debug a class, the programmer had to link it statically with IK to make the symbols of the class appear in the `ik` binary. We then implemented a solution similar to PCR [Weiser 89], in which the `ik` symbol table is updated each time class code is loaded. Although this approach allows the use of standard debuggers, it does not provide a usual debugging environment, because class symbols remain unknown until the class is really linked and mapped in the application virtual space. This can be solved by changing the debugger to load classes when their symbols are requested in the debugging session.

The advantages of a run-time binding mechanism can also be explored for debugging purposes. The reactiveness provided by the system significantly reduces the compilation time and allows the replacement of a class mapped in an executing application, on the fly, without re-linking or even restarting the application. In practice, programmers use "on the fly" update of classes to test and correct small changes, and static linkage to introduce large modifications. After fixing problems in statically linked class code, there is no need to relink the whole application. Instead, a run-time option informs IK to ignore the image of a statically linked class, and dynamically link a newer version.

5.3 Transparency in the Model

The design of the IK platform was heavily influenced by the desire to provide a single uniform programming paradigm. Programming volatile, persistent, and distributed objects should be transparent to the developer. The concept of hiding from programmers all non relevant details, thus leaving only high level abstractions, is widely used. The question is to what extent this can be done without sacrificing other attributes, particularly performance.

Distributed applications based on client-server interaction expose distribution by distinguishing clients from servers. Our model does not have this distinction; the programmer is induced to forget about distribution. The performance of applications developed ignoring completely distribution aspects is dramatically dependent of the location of frequently used objects. If objects are on disk, they will be mapped in the application context and the application runs fast. If objects are already mapped elsewhere, the application runs slowly due to the large number of remote invocations. On the contrary, applications designed with a global state and a local replica of most accessed objects are able to compute with a reduced number of remote invocations. They run with acceptable performance and are not dependent of the initial location of the objects.

Distribution must be taken into account in the design phase of applications. However, we also learned the advantages of the IK uniform programming paradigm. Firstly, it allows a much faster prototyping of applications. Secondly, by keeping the code for object placement outside the applications, applications can be reconfigured to a wider spectrum of real life situations. Finally, combined with the object oriented nature of applications, it also allows a later restructuring of the applications to improve performance or robustness with minor effort.

The experience concerning persistence transparency is similar to distribution but less performance critical. Although performance can be improved by a careful structuring of applications, it is normally acceptable when applications are well structured.

The major problems occur when a considerable amount of transient data is saved to disk. This could happen when the instance data of stored objects includes a significant percentage of transient variables. However, we measured that even when applications were not designed with persistence in mind, in average only 2% of the instance data variables of persistent objects were transient. We are thus induced to conclude that good application design and structuring tends to separate persistent and transient data in different objects. Another situation were transient data can be unnecessarily saved, occurs when transient objects are referred to by persistent ones. In such cases, programmers must implement the `beforeUnmap/afterMap` upcalls in the classes to add or remove information from the graph[2].

We use persistence to keep the applications state between runs in data structures whose size is normally less that 0.4 Mbytes. Applications often flush their state to improve their robustness. However, they are not disk intensive applications, and the conclusion that persistence is not as critical as distribution during the application design phase cannot be generalized to other types of applications.

Persistence transparency is also supported by automatic object clustering based on locality and usage criteria. However, programmers cannot delete unused clusters, as it would be possible in a programmer defined clustering policy. A distributed garbage collector of clusters is an essential component of the IK platform. We are working on a distributed garbage collector of clusters derived from a mark and sweep algorithm with timing decision.

6. *Related Work*

Distributed operating system projects have progressed into object orientation by maintaining the notion of object within the system and implementing mechanisms for object invocation and object instantiation.

The computational model of Clouds [Dasgupta 88] is similar to the model adopted in IK despite Clouds only provides support for coarse-grained persistent objects. Clouds implements a kernel on top of bare

[2] For example, in figure 2 the `beforeUnmap` method prevents the lock object to be written to disk, and the `afterMap` method initializes the *lock* reference to a new lock object.

hardware, whereas the main stream of our project was focused on providing a platform on top of existing UNIX systems.

Eden [Black 85] also deals with coarse-grained objects, but they are active. Each object has a thread responsible for incoming invocations. In IK threads are not attached to objects. Capabilities were used for object identification and protection, but required objects to have a private hardware protected address space.

The previous systems, as well as some others (e.g. Argus [Liskov 87] and Hermes [Black 89]) do not provide a uniform object model, because the objects they support are too large to be the building blocks of applications. A different approach has been followed by other systems, providing a uniform object model, directly supporting fine-grained objects.

The lessons from Emerald [Black 86] were particularly important in the definition of our computational model, since Emerald also provides a uniform paradigm. A significant different is that, in Emerald, types are classified at compile time and all objects of the same type are either global or local. In IK objects are promoted individually to global objects at run-time. Emerald exploits the call-by-move parameter passing mode. However, much of the work is left to the Emerald compiler as it must decide which objects are passed in the call-by-move mode. We wanted to keep the EC++ compiler simpler and, at the same time, keep the decisions independent of the language.

Like IK, Amadeus [Cahill 90] was developed within the Comandos project. Although sharing the same model and general architecture, they differ at both the language and implementation level. New keywords were added to the language in Amadeus to express persistence and distribution, whereas we decided to use inheritance and retain the original language syntax.

A different approach to distribution is to use objects to support the client-server model. This was the path followed by Extended-C++ [Seliger 90], by extending the C++ to program client and servers. The existence of remote objects is explicitly defined by the programmer, using the keyword *remotable* in classes and pointers. The concept of global reference is not supported, preventing objects to be passed by reference and also the invocation of objects not registered in the Name Server. There is no concept of persistence or automatic activation of objects; if a remote object ends its execution permanently or temporarily, all remote invocations to it will fail.

The SOS [Shapiro 89] project developed an object kernel to handle distribution and persistence based on the proxy mechanism. In SOS, object invocation outside a context must be preceded by explicit importation of a proxy from the server. The server may decide to export itself, or to delegate part of its functionality to the client node. SOS does not hide distribution and persistence from the programmer, making proxy imports visible to clients, and exports visible to servers.

7. Conclusions

IK has become an exploitable platform upon which some experiences can be conducted. We are now defining a large project for a business integrated system where the technology of IK will be useful to develop applications for distributed decision support, distributed document handling and contextual communication between groups. However, we still lack some mechanisms for exception handling and protection which we hope to develop in parallel.

Clustering techniques have proven to be essential in reducing the granularity mismatch between language level objects and the files manipulated by persistent storage. The persistence provided by the platform does not offer the same level of functionality found in traditional databases; there is no support for queries nor transactions. Nevertheless, we have substituted the use of databases by persistent objects in a couple of applications were query processing is fairly limited (Type Manager and Browser) with significant performance improvements.

Another important objective, not easily quantifiable, is the easiness of use of the platform. C++ was a choice dictated by the industrial weight on this language. However, the non object oriented features in the language are extremely difficult to circumvent. Clearly trying to be as compatible as possible with C++ accounts for a large part of our effort and, although the restrictions imposed are non intrusive for people developing new applications, they are a burden when porting existing code. Eliminating the restrictions completely is complex and can lead to situations where normal C++ shortcuts perform poorly. No obvious alternative exists as there are no other languages with the acceptance of C++.

The programming environment is now quite usable, although debug still needs to be better integrated with distribution and dynamic linking. We are also developing a simple CASE tool to guide programmers in the design and structure of applications, taking into account distribution and persistence.

Acknowledgements

We would like to thank Helena Oliveira, David Matos, Ricardo Nunes, António Rito and specially João Pereira for bearing with early IK implementations when developing their applications; and Nuno Neves, Pedro Trancoso, Miguel Castro, Adriano Couto and Paulo Meneses for their work on the IK platform. We also thank Peter Dickman, Roger Lea, David Harper and Nuno Guimarães for their helpful suggestions during the preparation of this document.

References

[Atkinson 83] M. P. Atkinson, P. J. Cockshott, K. J. Chisholm, and R. Morrison. An Approach to Persistent Programming. *Computer Journal,* 26(4):360–365, November/December 1983.

[Black 85] Andrew P. Black. Supporting Distributed Applications: Experience with Eden. In *10th ACM Symposium on Operating Systems Principles,* pages 181–193, Orcas Island, Washington U.S.A., December 1985. SIGOPS and ACM.

[Black 86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *OOPSLA '86 Proceedings,* pages 78–86, Portland, Oregon, September 1986.

[Black 89] Andrew P. Black and Yueshayahu Artsy. Implementing Location Independent Invocation. In *Proceedings of the 9th International Conference on Distributed Computing Systems,* pages 550–558, 1989.

[Cahill 90] Vinny Cahill, Chris Horn, Andre Kramer, Maurice Martin, and Gradimir Starovic. C** and Eiffel**: Languages for Distribution and Persistence. In *OSF Microkernel Applications Workshop,* Grenoble, France, 1990.

[Castro 93] Miguel Castro, Nuno Neves, Pedro Trancoso, and Pedro Sousa. MIKE: a Distributed Object-oriented Programming Platform on top of the Mach Micro-Kernel. In *Proceedings of the USENIX Mach Conference,* Santa Fé, April 1993.

[Chase 89] J. Chase, F. Amador, H. Levy, and R. Littlefield. The Amber System: Parallel programming on a network of multiprocessors. In *Proc 12th ACM Symposium on Operating Systems Principles,* Litchfield Park, USA, December 1989.

[Cox 86] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[Dasgupta 88] Partha Dasgupta, Richard Leblanc, and William Appelbe. The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work. In *8th International Conference on Distributed Computing Systems,* pages 2–9, S. José CA (USA), June 1988. IEEE.

[Decouchant 88] D. Decouchant, A. Duda, A. Freyssinet, M. Riveill, X. Rousset de Pina, R. Scioville, and G. Vandôme. GUIDE: An Implementation of the COMANDOS Object-Oriented Distributed Architecture on UNIX. In *Proceedings of EUUG Conference,* Lisbon (Portugal), October 1988.

[Ferreira 91] Paulo Ferreira. Reclaiming Storage in an Object Oriented Platform Supporting Extended C++ and Objective-C Applications. In *Proceedings of the International Workshop on Object Orientation in Operating Systems - IEEE,* Palo-Alto, October 1991.

[Fowler 85] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, Seattle, December 1985.

[Gamma 88] Erich Gamma, André Weinand, and Rudolf Marty. ET++ - An Object-Oriented Application Framework in C++. In *EUUG,* Cascais, October 1988.

[Goldberg 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Kaehler 86] Ted Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *Proceedings of OOPSLA 86,* Portland, Oregon, September 1986.

[Krakowiak 90] S. Krakowiak, M. Meysemburg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *JOOP,* pages 11–21, September/October 1990.

[Liskov 87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. *Proceedings of the 11th ACM Symposium on Operating Systems Principles,* November 1987.

[Marques 88] José Alves Marques, Roland Balter, Vinny Cahill, Paulo Guedes, Neville Harris, Chris Horn, Sacha Krakoviak, Andre Kramer, John Slattery, and Gerard Vandome. Implementing the COMANDOS Architecture. In *Proceedings of Esprit Technical Week,* Brussels (Belgium), November 1988. North-Holland.

[Marques 89] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA 89,* New Orleans, 2-6th October 1989.

[Ope91] Open Software Foundation, Grenoble. *DCE Application Development Guide,* March 1991.

[Richardson 89] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and Implementation. *Software-Practice and Experience,* 19(12), December 1989.

[Seliger 90] Robert Seliger. Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling and Garbage Collection. In *USENIX C++ Conference,* pages 241–263, 1990.

[Sequeira 91] Manuel Sequeira and José Alves Marques. Can C++ be Used for Programming Distributed and Persistent Objects? In *Proceedings of the International Workshop on Object Orientation in Operating Systems-IEEE,* Palo Alto, October 1991.

[Shapiro 89] Marc Shapiro. Prototyping a Distributed Object-Oriented Operating System on Unix. In *Distributed and Multiprocessor Systems Workshop,* Fort Lauderdale, FL, October 5-6 1989.

[Sun86] Sun Microsystems. *External Data Representation Protocol Specification,* February 1986. Revision B.

[Ungar 84] David Ungar. Generation Scavenging: A Non-disruptive High Perfomance Storage Reclamation Algorithm. In *ACM Software Engineering Symposium on Practical Software Developments Environments,* pages 157–167, Pittsburgh, April 1984.

[Weiser 89] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime Approach to Interoperability. Technical report, Xerox PARC, March 1989.

[Wilson 91] Paul R. Wilson. Operating System Support for Small Objects. In *Proceedings of International Workshop on Object Orientation in Operating Systems,* October 17-18 1991.

[submitted Feb. 1, 1993; revised May 10, 1993; accepted June 2, 1993]