1 Abstract

This article describes an Object eXchange Service (OXS) and an EXTernal Object Representation (EXTOR) in the context of a distributed object-oriented database. EXTOR is the common representation which facilitates the sharing of information among a network of machines. This work is guided by the desire to represent complex structured information in an efficient external form and to provide maximal sharing of information among computing sessions. OXS is a service which, together with the concepts of object boundary, global objects, and object closure, provides the exchange of information via EXTOR. The article also contains a brief description of the distributed object-oriented system which utilizes this service, the performance results of OXS, and the future directions for the design of generalized object exchange services in heterogeneous computing environments. The work described here is part of Zeitgeist, a Distributed Object-Oriented Database system which is being developed at the Computer Science Center of Texas Instruments, Dallas.

2 Introduction

A major goal of the Zeitgeist project[8] is to provide a programming environment where a programmer can manipulate persistent and transient data in the same manner. Furthermore, transactions can be implemented on persistent objects to support cooperative design applications. This goal is achieved by providing a seamless integration of an object-oriented database with an object-oriented programming environment, namely Common Lisp with Flavors on a network of Texas Instruments Explorer workstations. A key component of our system (as shown in Figure 2.1) is the translation of an object into a representation which can be easily stored onto and retrieved from disk. The representation of an object on disk is referred to as an EXTOR. The EXTOR of any object is an array of 16 bit elements, which is essentially a linearized representation of the object in virtual memory with encoding of type information. An EXTOR can embed references to other EXTORs by means of long pointers. The embedded references are necessary to preserve the correct semantics of shared objects. A long pointer is a system-wide unique identifier in time as well as in space; the long pointer mechanism expands the address space of the machine virtually without limits. An object management system, with the help of the Object eXchange Service (OXS), transparently resolves the long pointer references. The loading of objects from disk, on demand, to resolve such references is called Object Faulting.

Common Lisp systems on conventional machines provide two ways of making objects persistent. One can make objects persistent by saving the entire Lisp environment to a disk band. This is a memory dump and the user has no control over which objects are saved; also there is no way in which objects made persistent under two separate save operations can interact. There are cases where this is the right facility, namely when we are creating an operating environment mostly consisting of programs. A second way of making objects persistent is to use the fast dump facility[7]; in this case, individual objects can be made persistent and retrieved by recreating the data when disk representations are loaded into memory. The main drawback of this facility is the inability to provide inter-object references for sharing purposes. Our external representation is designed to preserve the structure of the object's memory image form. One of the design goals for external representation is to provide isomorphism between in-memory and external representations of objects for the reason of preserving the type information of objects. Due to this memory image form, there is little performance penalty involved in recreating an object from its EXTOR. The design of EXTOR combines the reference flexibility of memory dumps and the object selectivity of fast dump. In addition, it supports the object faulting mechanism for demand loading and address space expansion.

The work described in this article is guided by the desire...
to represent complex structured information in an efficient external form and to provide maximal sharing of information among computing sessions. We discuss EXTOR and OXS and briefly touch on the Object Management System (OMS) which makes use of all the features of OXS in building a distributed OODB. A detailed report on the Zeitgeist project appears in [8].

Section 3 of the article introduces the problems associated with external representation. Section 4 briefly describes related work done in the area of external representation. Section 5 describes EXTOR with the help of an example. OXS and its design concepts are presented in section 6 and 7. Section 8 briefly describes the implementation of OXS and provides early performance results. Finally, section 9 discusses directions for further research.

3 Problems associated with external representation

There are several problems associated with the creation of the external representations of objects. One of the most obvious is that of object boundary. In Lisp systems (like in many other systems), the objects in memory are highly interconnected with the system's run-time environment. Simple tree traversal based translation of an object graph in memory could lead to visiting all the objects in memory.

The concept of object boundary, which is discussed in section 6 of this paper, solves this problem.

The second problem is caused by the desire to share objects among various users of a distributed system. Since the objects are being shared across machine boundaries, a mechanism for identifying shared objects and primitives for referencing these objects becomes necessary. Section 6 of this article discusses an object boundary mechanism which addresses the identification of shared objects and Section 5 discusses a long pointer mechanism which provides referencing primitives for these objects.

The third problem in creating external representations relates to the global objects of the run-time environment. Objects like NIL and T are universally available in all Lisp systems. Translation of these objects is unnecessary and the concepts introduced above are insufficient to deal with objects of this kind. In Section 6, the concept of global objects is introduced to address this issue.

Finally there is the critical issue of performance in creating external representations. The creation of external representations for large numbers of small objects and the transportation of them across machine boundaries creates a performance bottleneck. The concept of object closure, introduced in section 6, provides a step towards solving this problem.
4 Related work in external representation

There are several approaches to translating the computational form of an object into a form that can be saved and later translated back into the computational form. A general approach is to translate the object into a structured text. This is the approach taken, for example, in the Electronic Data Interchange Format standard[6]. This approach has the advantage of genericity, but it suffers from performance problems and excess storage due to object size expansion. The appearance of portability is misleading; even though the external form is readable on any machine, a different filter is needed for each machine to translate the data back to its computational form.

XDR from Sun Microsystems[14] addresses the performance problem to a large extent. The XDR standard provides library routines for deriving the external representation of primitive data types. More complex data are built using the primitive data types and user written read and write routines. Portability is achieved using various filters. XDR facilitates the transmission of data between machines having different word types and different representations for primitive types like numbers. XDR's approach of linearizing data structures is similar to our approach. However, inter-object referencing and object faulting are critical to our design, and XDR leaves it up to the user to provide these capabilities. XDR’s emphasis on providing primitive data types, and our primary goal of providing a persistent world (with demand-driven incremental object retrieve/save) which effectively parallels the programmer's workspace of complex objects, contributes to the divergence of the designs.

University efforts, such as the Mach/Matcher system at Carnegie-Mellon University[10] and the Saguaro project at the University of Arizona at Tucson[1] are driven by the need to integrate the heterogeneous environments of languages and computers. These efforts are directed towards designing an object specification language and an external representation. However, their external representations can handle only the primitive data types of the programming languages and lack the ability to handle composite structures with pointers and abstract data types. In the general programming community, there has been interesting research aimed at eliminating the impedance mismatch between programming languages and databases[2], [5], [11]. Researchers have also pursued the ideas of uniform memory abstraction[12] and extended address space[4], [9].

The Orion object-oriented database project at MCC[3] and Statice at Symbolics[13] have addressed issues similar to the ones discussed here.

Existing approaches delegate responsibility for representing complex structures to the user; they provide no mechanisms for specifying object boundaries and inter-object references. We believe that the EXTOR adequately addresses issues of complex object representation, object boundaries and inter-object references. Zeitgeist provides a seamless integration of programming language and database using EXTOR and OXS.

5 The EXternal Object Representation

This section describes the EXTOR with the help of an example. The external representation scheme, EXTOR, defines a collection of primitive data types, a collection of composite data types, and a set of object referencing primitives.

In object-oriented programming, the descriptions of data and operations are encapsulated in an entity called an object. In control-oriented programming, such encapsulation is not present. In both cases, we can think of the data in an object as a piece of typed information. The type information may be present directly in the object at run-time or may be derivable from the system's compile-time environment.

The EXTOR of an object consists of a set of data types and values. Some of the data types are primitive data types whose values are atomic and others are composite data types. Composite data types help build a complex object from primitive data types and constructors.

Primitive data types of the EXTOR include numbers and characters. Composite data types may consist of complex structures such as lists, symbols, arrays, records, functions, and abstract data types (object instances and object handlers[7]).

In addition to these two classes of data types, EXTOR supports relative references and named references, which allow sharing within an object and among different objects.

Portability is achieved using various filters. XDR facilitates the transmission of data between machines having different word types and different representations for primitive types like numbers. XDR’s emphasis on providing primitive data types, and our primary goal of providing a persistent world (with demand-driven incremental object retrieve/save) which effectively parallels the programmer's workspace of complex objects, contributes to the divergence of the designs.

Let us consider the external representation of an array object in further detail. Figure 5.1 shows an example of an array object which contains information about the city of Dallas. It has three elements: the first is a character (a primitive data type), the second is a list of two elements (a composite data type of city name, DAL, and, state name, TX), and the third is an object of user-defined type (airport object). The external representation of this array object is shown in Figure 5.2. The 0th word of the EXTOR defines the header. The total size of the EXTOR is 14 words. The 1st word contains the header information of the array object and word 2 to 4 contain objects (or their relative references) pointed to by the array object. The character object is in the 2nd word. The list object in the 3rd word and airport object at 4th word contain relative references. The two elements of the list are represented in the 5th and 6th words, which have relative references in them for the real string objects in the 7th and 8th words. In this example, the user-defined airport object forms the boundary (a concept described in the following section) for the array object. In the 11th word, where the airport object should be represented, a long pointer identifying the airport object has been placed in the EXTOR.
6 Design Concepts of Object Exchange Service

This section describes three concepts, Global Objects, Object Boundary, and Object Closure, which form the kernel of the Object Exchange Service (OXS).

6.1 Global Object

Global Objects are those objects which are always present in the environment, or can be created on demand by the Object Management System (OMS). Examples of global objects are the primitive objects nil and true and various font map encodings. It is not necessary to save global objects to the disk. In some cases, e.g. the display screen, saving and retrieving a global object may create consistency problems. Applications, in consultation with OMS, decide which objects are global. OMS manages a catalogue of global objects and is responsible for generating and resolving externally referenced data for OXS. For global objects, OXS encodes an external reference of type Global Object. A transient object referenced from a persistent object but whose value is uninteresting may be encoded as the global object nil. When translating from an external representation, OXS resolves global object references by asking OMS to return the corresponding object from its catalogue. If OMS cannot find or create the object, it may consult the user or generate an error.

The concept of global objects helps in isolating the content of an object from its environment. The contention is that the environment, or a large portion of it, will exist because of the presence of the application or of the software platform on which the application runs. Our experience with Computer Aided Design applications bears this out and indicates that the conscious use of global objects can significantly reduce OXS overhead.
6.2 Object Boundary

The Object Boundary determines the granularity of the objects which are individually handled for system facilities like object transport, object locking, and clustering. It is quite conceivable that the system may provide differing levels of granularity for some of these facilities. For example, transporting an automobile object in a design environment conceptually moves everything in the closure of the automobile object, but, in order to reduce access-time to the root object, or to permit sharing of components (e.g. engines) between designs, some sub-objects may be defined as separate referenced by long pointers. OXS does not allow accessing a partial disk object or locking a partial disk object; therefore, object boundary determines the lowest level of granularity addressable in the system.

Virtual memory objects reference other objects by pointers. In saving an object to disk, we need to save all the objects which are also referenced by the object, namely the transitive closure. The transitive closure may consist of global objects, shared objects (objects which are also in the transitive closure of another object) and other objects. The global objects and the shared objects should be handled as external references for semantic reasons. Other objects in the transitive closure may also be handled via external reference. The object boundary consists of all objects in the transitive closure which are referenced by an external reference in the EXTOR. It determines where an object ends and where another object begins. All objects within the boundary of an object are separately persistent and can be transported, locked, and accessed independent of other objects. However, OMS does not guarantee that these objects have handles which are available to users. In EXTOR, a reference to an object within the boundary of an object is a relative reference and a reference which crosses the object boundary is an external reference. As shown in Figure 5.1, although OBJ2 is in the transitive closure of OBJ1, it is not within OBJ1. OBJ1 contains OBJ3 and OBJ4. These objects are represented in EXTOR as they are; whereas for OBJ2 only the corresponding external reference is placed. In OXS, we write large objects as separately persistent objects basically for network performance reasons. Also, all Lisp symbols are made separately persistent, based on the knowledge that they are often shared.

6.3 Object Closure

Object Closure is important in deciding how much should be restored when the user requests an object. Is it better to restore just the referenced object (i.e. the 1st level closure), or should the objects it references in the transitive closure be restored as well? To answer this question, the object closure level, an integer passed to OXS, is used. An object closure level of zero implies that all the objects in the transitive closure shall be restored at once.

The default policy is to use first level closure. The referenced objects are brought in on demand by object faulting. The closure level specification is a performance tuning parameter and basically provides the system control over the amount of pre-loading of data. When an EXTOR is developed, the layering of the object into different levels results from embedding long pointers at strategic points. Users can control this layering by giving directives to the Object Management System. The default layering we have implemented in Zeitgeist's OMS attempts to guarantee proper semantics for shared objects and increased efficiency at the storage level.

7 Object eXchange Service

Figure 2.1 shows the various components of Zeitgeist. Applications interact with the objects through the Object Management System (OMS). The Object Exchange Service (OXS) has little knowledge of the application environment which exists in the programmers' workspace. OXS queries OMS to decide which objects are shared objects and which objects are global objects; boundaries of objects are also determined by OMS. OMS makes these decisions based on a set of rules. By default, OMS has a list of global objects and a set of rules to determine sharing. Applications add or remove rules to this default set to customize the application environment. In the case of abstract data types, this may be done by inheriting a shared-object attribute flag. Partial instances of the abstract data type may be marked as shared by turning on the flag. More typically, all instances of the type may be marked as shared by defaulting this flag to be on when an instance is created. The list of global objects is also modified by the application: individual objects or collection of objects (e.g. Common Lisp Packages) may be declared as global. For space and time efficiency, it is ideal to determine the minimal set of shared objects and the maximal set of global objects for an application. This is not practical in most cases. We believe that the decision to delegate these determinations to OMS and ultimately to the user is the right approach.

When translating an object into external form, OXS first determines the boundary of the object. An object is bounded by the global objects and the shared objects it references. The boundary determines which references are external references and which references are relative references. With the help of an object definition dictionary and a machine representation dictionary, objects are encoded into EXTOR by translation routines in OXS. The object definition dictionary is a collection of the type definitions of the objects. The machine representation dictionary defines the virtual machine for the hardware on which objects have been implemented. External references are encoded as long pointers and relative references are encoded as byte offsets. External reference may be of two types, Global Objects and Non-Global Objects. OXS, via OMS, interacts with a multi-level name server to generate and manage the long pointers used by EXTOR. Header words are added to indicate properties of the external representation, (i.e. an identifier for the routine to be used to restore the object.
OBJECT SIZE (32-BIT WORDS)

FIG. 7.1 - PERFORMANCE OF OXS

The OXS approach closely resembles Bishop's object copying algorithm for area-wide garbage collection in virtual memories, except that in OXS the object boundary concept with global objects provides a stopping point for the transitive traversal of an object. Furthermore, in our scheme, storage devices and/or distributed systems appear to be extensions of the virtual memory of a computing system.

When an object is brought back from persistent storage, OXS translates the EXTOR into an internal representation, possibly using some information stored in the header words. During this translation, relative references are replaced by virtual memory pointers. When an external reference (long pointer) is encountered, OXS determines, based on type, whether the long pointer references a global object. In this case, the long pointer is replaced by a virtual memory pointer to the referenced global object. Other external references are replaced by encapsulations. Encapsulations are OMS data structures which are surrogates for the actual objects on disk. The surrogates implement an indirection; when an application reference resolves to a surrogate, the actual object is brought from disk and returned to the application in a transparent fashion (object faulting). Alternately, users can request that objects are fetched in an eager fashion to avoid runtime faulting; the surrogates are still set up so that the sharing semantics can be derived when the objects are written back to the disk. It is also possible to specify that eager fetching take place up to a certain level beyond which objects are faulted in. Object characteristics contained in the header words of the EXTOR (e.g. read only status) are passed to OMS.

8 Implementation and performance results

OXS is built around a dictionary of translation routines for the various types of objects in the Lisp environment. A depth-first search algorithm is used to traverse the nodes of the object graph in memory. Depending on the type of a node, a specific routine is invoked to linearize the node into EXTOR form. Each node is examined to determine whether or not it should be within the boundary of the object being translated. If outside, then a long pointer corresponding to the object is found or created. Global objects are handled in a similar manner. When retrieving an object from disk, the reverse process is executed. Each long pointer is converted to a physical pointer if the object corresponding to the long pointer is already in memory, otherwise the object fault mechanism is set up to bring the object from disk when it is referenced.

We have not yet optimized EXTOR and OXS for perfor-
ment; however the following is a brief summary of our preliminary observations. In general, OXS translation time is proportional to the size of the Common Lisp object and its transitive closure being translated. The size of an EXTOR is about the same as the Common Lisp object it represents. Figure 7.1 graphs object size versus translation time and EXTOR size for a representative set of Common Lisp objects. All measurements were taken on a Texas Instruments Explorer II. Notice that there is currently about a 0.15 second overhead per top-level translation, which makes small objects more expensive to translate.

For flat or immediate objects, restoration times are comparable to save times, but for objects with sub-objects that have been made separately persistent, significant reductions in access time can be realized by avoiding restoration of the sub-objects before they are needed. For example, OXS translation time for an array of 100 1Kx1K bitmaps is 109.391 seconds, 3.405 seconds for the 100 element array, and 1.060 seconds for each of its bitmaps. Restoration of this array takes 0.18 seconds of OXS translation time because only the 100 element array is restored; the bitmaps are not. The object faulting mechanism takes care of restoring the bitmaps if and when needed. The execution of the standard Common Lisp array accessor causes the first bitmap to be restored and returns it as a value. The OXS translation of the bitmap adds 0.95 seconds to the normal execution time of the array reference, but only the first time. Subsequent accesses of the same element have no overhead. If two slots point to the same persistent element, then the first slot referenced will cause the object to be restored, and the second will incur a very small overhead to resolve the reference. Furthermore, subsequent saves of the object also benefit. If the array were to be saved again, bitmaps that had not yet been faulted in would not be saved.

We plan to speed up translation, improve the heuristics for the OMS, and add sparse data compression protocols to the EXTOR to reduce space in the persistent store and improve speed on restoration.

9 Conclusions and directions for future research work

EXTOR provides ability to reference the elements of an object via relative pointers and to reference other objects using embedded long pointers. Relative references allow structured information to be conveniently represented in the external form. This can be used to build a composite object from primitive objects. Long pointers provide correct semantics for shared objects and also expand the address space of a machine limitlessly. EXTOR, being very close to native representation, is compact and efficient for translating from/to native form. We have designed the external representation to be open ended; new types or classes can be easily added if the language is enhanced. This feature will be utilized in making our system cognizant of CLOS.

EXTOR can also be used for the distribution of processes among the computing elements of a distributed/parallel computing environment. Further research is required, however, to comprehend the abstraction of processes as data structures. Once this is done, EXTOR can be used in parallel/distributed systems to transfer processes or save processes for continuing computation later.

EXTOR and OXS are used extensively in a distributed OODB for VLSI design application at Texas Instruments. VLSI designs are data intensive that most designs will not fit all at one time in the virtual memory of a work station. The ability of EXTOR to handle complex structures and to provide support for the fetching of objects on demand make our system a natural fit for handling large designs. Design libraries are implemented using the notion of read-only objects. Database support, in the form of concurrency control, recovery, and transaction management are provided by other components of Zeitgeist.

Finally, it is our intention that OXS evolve into a truly generic object exchange service, providing language and hardware independent support to a variety of clients for the exchange, storage and retrieval of structured information and unstructured data. At present, we are examining current work, especially in the fields of distributed and parallel computing, as we put together a design that is flexible and expressive enough to support at least Common Lisp, C++, and Ada implementations on different hardware and software platforms, and is efficient enough to minimize storage space, communication time, and translation overheads. In many respects, these goals are conflicting. Our current design favors efficiency in Common Lisp on Lisp machines over flexibility, because our customers wouldn't use a low-performance system, no matter how elegant. If we can provide an efficient common external object representation for a representative set of programming languages, then we can avoid providing a multitude of translation routines from one language-specific representation to another. This would be a combinatorially explosive task as new languages are added, and one which depends for success on minimal object movement between heterogeneous environments, a situation we expect will not be the norm, especially in the CAD environment we work in. It is our objective, therefore, to design and implement a truly heterogeneous object exchange service centered around a generic external object representation.

The Common Lisp Object System, an object-oriented extension to Common Lisp, has been adopted by the X3J13 committee, which is defining ANSI Standard Common Lisp.
References


