OBJECT DIRECTORY DESIGN FOR A FULLY DISTRIBUTED PERSISTENT OBJECT SYSTEM

John A. Mathew, Peter C.J. Graham, and Ken E. Barker
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada R3T 2N2
(204) 474-8313
{jmat,pgraham,barker}@cs.umanitoba.ca

ABSTRACT

In large persistent object systems it is possible to have many million (or eventually billion or more) objects in existence. These objects must be managed to enable their effective and convenient use. At the heart of this management problem is the structure which maintains information about the objects and provides services on the objects using the information it maintains. We refer to this structure as a *directory* of objects.

In a transparently distributed object system the directory must contain entries for all objects in the system regardless of their location. To ensure both efficiency and reliability any implementation of a *global* directory of objects (GDO) must be done carefully. This paper details the design of a GDO for a fully distributed persistent object system that is currently being built in a single shared address space. The design of the directory offers efficient, scalable, and reliable services across a distributed environment. Further, although it is designed for a specific (and somewhat unusual) environment, the implementation ideas are applicable to other distributed object systems.

1 Introduction

This paper addresses the issue of managing large collections of persistent objects in a distributed environment. It does this by focusing on the key data structure required for object management, the object directory. Logically a single, global directory of objects (GDO) is desired but this is not a practical implementation strategy. Instead, a carefully distributed data structure is required that provides reliable, efficient, and scalable services while maintaining the appearance of a single directory to its users.

The GDO for a persistent object system acts, *passively*, as a repository for information about the objects being maintained in the system. It also acts, *actively*, as a provider of management services on those same objects. Depending on the goals of a given implementation, either view may be emphasized or de-emphasized to provide object management mechanisms of significantly different flavour but essentially the same capabilities.

The GDO structure described in this paper is for a prototype distributed object system [GB93, GBBZ93, BPG95, GBPZ95] which is currently being implemented in a shared persistent address space. While this environment does introduce some unique issues to the problem of object management, it does not preclude the more general application of the techniques described herein to other object systems. The focus of this paper is the GDO as it relates specifically to object management. In general, a GDO may be used for more than just this purpose. Additional possible uses of a GDO will be discussed later in the paper as future work.

This rest of this paper is organized as follows. In Section 2 we discuss our assumed environment and the motivation behind our work. Section 3 gives a high level discussion of what a GDO is and what its requirements are. An evaluation of possible GDO designs implemented using conventional approaches to building distributed structures is presented in Section 4. We then present our GDO design, a combination of existing design techniques, in Section 5 and evaluate it in Section 6. Related work is described in Section 7. Finally, Section 8 makes some concluding remarks and proposes directions for future research.

2 Environment and Motivation

The assumed environment and motivation may seem to be unusual topics to group together in a single section. They are discussed together in this paper because the nature of our development environment, in part, affects the motivation behind our design. Further, motivation for the environment itself must be briefly discussed.

2.1 Environment

This work has evolved out of the ongoing DSVM (Distributed Shared Virtual Memory) project [GB93, GBBZ93, BPG95, GBPZ95] and as such is oriented towards a transparently distributed persistent object system. To place this work in perspective to other distributed object systems, a brief overview of the DSVM environment is now presented. For the sake of brevity, many details are omitted while others, those relevant to a discussion of the GDO, are emphasized.

The DSVM project seeks to build a distributed persistent object programming environment in a single, 64 bit, shared address space. Utilizing, evolving technology including 64 bit processor architectures, it is now feasible to build an efficient persistent and distributed shared memory. Our interest in doing this is to place objects into the persistent memory and thereby have them shared and transparently accessible across a collection of interconnected machines constituting a distributed system. Numerous advantages are achieved when building a persistent object system in this way. These include simplified programming, since both distribution and persistence are provided transparently, and certain implementation benefits including eliminating the need to swizzle [WK92, WD92, KK93] object references. This is because in a 64 bit DSVM system object references are simply persistent virtual addresses.

We propose to use DSVM as a platform for research in interoperability [BPG95] using persistent object technology. A very high level diagram of the proposed *logical* system architecture showing the involvement of the GDO, is given in Figure 1. A high level *physical* view of a DSVM system is also provided in Figure 2.

For the purpose of this paper, the interoperability issues are not a fundamental concern. Thus, the relevant portion of Figure 1 is from the transaction manager interface down. Working from the bottom up, our DSVM system for persistent objects provides a distribution mechanism (shown in Figure 2), persistent (on disk) storage, virtual memory management (to effect the shared memory illusion), transaction support (for simplified programming), and an object information repository (the GDO). With the exception of the distribution mechanism, these components are all shown in Figure 1.

In addition to understanding the basics of the DSVM design approach, it is also important to understand our assumed object model (including the execution model). For the purpose of this discussion, the object model may be assumed to be "vanilla flavoured". The core object concepts are supported (ADTs with strict encapsulation, plus inheritance and polymorphism) but more advanced features such as meta information [1] are ignored. This does not preclude their introduction into the DSVM system but merely reflects their lack of direct relevance to the discussion at hand.

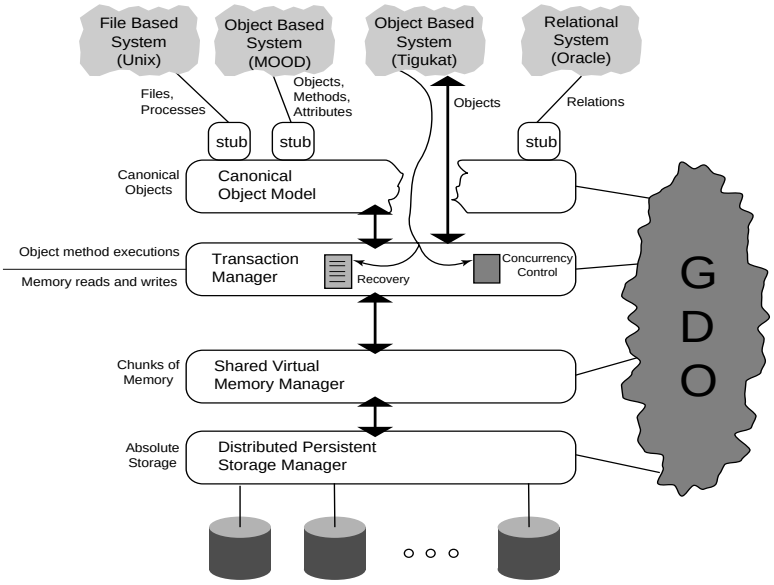

Figure 1: Interoperability System Architecture

Objects, as discussed in this paper, consist of state (attributes) and behaviour (methods). Further, each object has its own unique object identifier (OID) which, in our DSVM system, is the object's virtual address in the *object space* where it is persistently stored. Each object is instantiated from a *type* specification which details the structure of its attributes and the implementation of its methods. All objects instantiated from a given type belong to the same *class*. Inheritance between types (and classes) is supported so that the characteristics of one *subtype* may be specified as a modification of the characteristics of a related *supertype*. Further, polymorphism is supported as an object of some class C may always be treated as an object of another class C' as long as C' is a superclass of C.

Method executions, whether by users or other objects, are treated as atomic transactions and a closed nested transaction hierarchy [Mos85, Wei89] results. Transactional execution is used to permit guarantees with respect to concurrency control and reliability where they are necessary. Equally importantly, transactions remove much of the complexity of distributed programming whether or not the distribution is transparent.

An object appears in the shared object space as a sequence of three components:

[1] Much of the information stored in the GDO is actually meta information for the corresponding object(s). Ultimately, the GDO will be part of an implementation structure for meta-object support in a *uniform* DSVM-based object system.

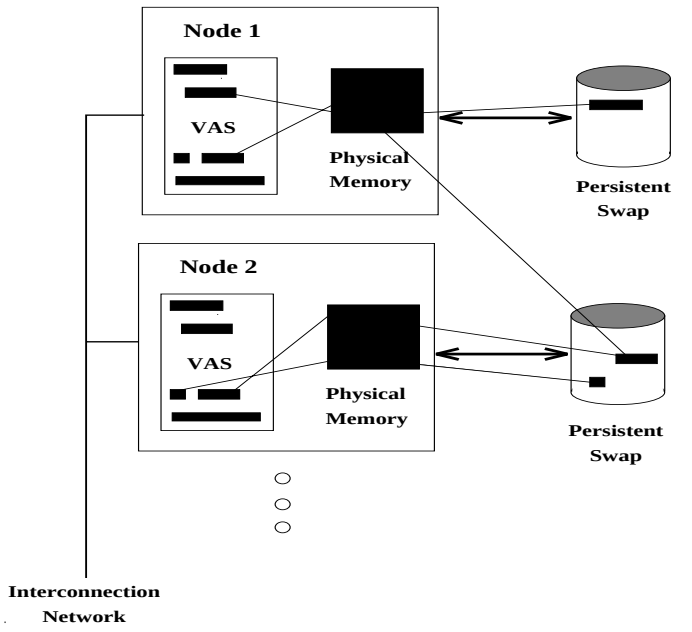

Figure 2: Physical System Structure

1. An indexed jump instruction which invokes the appropriate object method based on the value of a "method identifier" provided by the invoker of a method on the object[2],
2. The attributes of the object (in internal machine format), and
3. The method code for the object (in machine code form).

The mechanism enabling management functions to be enforced on objects in a shared address space is based on detecting object references. This is done by exerting explicit control over certain aspects of memory management. Referring to Figure 2, at boot time both the physical memory and the virtual address space (VAS) of any given processor (as represented by the page map table for the processor) are empty but the GDO is accessible at some well known location. Thus, when a transaction invokes a method (M_i) on some object (O^j), a reference to the virtual address corresponding to O^j is generated. This results in a *mapping fault* (commonly called a segmentation fault) and the system gains control. The GDO is then consulted using the virtual address of O^j as a key for the lookup[3]. If an entry is found in the GDO then information contained therein is used to build page table entries in the current processor for object O^j. If an entry is not found, then an invalid object reference has been detected. In either case, the mapping fault is resolved. The key to object management is that the mapping fault is detected and thus a point where authorization, object locking, and related functions may be performed is introduced without requiring that code be specially generated by the object compiler. Once an object is mapped, it may be referenced in memory with pages being provided via conventional demand paging.

It is also important to be able to detect the end of a method execution to perform other object management functions (such as the freeing of previously obtained locks). This too can be done without requiring modification to the object compilers used to create the method code and attribute templates for objects. This is accomplished by modifying the runtime stack when the method invocation is first made. The return address may be changed to point to a system-supplied routine for handling the end of a method invocation rather than the next instruction in the invoker of the method. The system supplied routine performs the necessary object management functions and then returns to the original invoker.

2.2 Motivation

It is clear from Figure 1 that the GDO is involved in all aspects of the distributed management of objects in a DSVM based persistent object system. Thus, its correct design and efficient and reliable implementation are key to the success of building such a system. Meeting all the desired goals of a GDO is a hard problem since many of the goals are contradictory.

The immediate goals of a GDO design and implementation are:

- flexibility,
- distributability,
- time efficiency,
- space efficiency,
- reliability, and
- scalability.

[2] The need for this initial jump instruction is due to an implementation "requirement" that all methods be invoked at offset 0 from the start of the object in the shared address space. This restriction can be relaxed but has implications on the practicality of making objects immutable.

[3] Since all methods are invoked via a single address at the beginning of an object's representation in the shared address space, key matching based on equivalence may be performed. This is one of two distinct benefits offered by having all method invocations pass through address zero relative to the start of the object.

A GDO must be designed to support flexibility since the requirements of object management may change as systems evolve and object models are extended. What is common to all systems that incorporate a GDO is the core concept of an object and that there is a single GDO entry per object. The contents of each entry, the number of such entries, and the relationships between entries must be as flexible as possible.

To be practical in a large scale distributed system, the GDO must be a distributable structure. This is necessary to support distribution of the workload associated with performing the object management functions based on the information stored in the GDO. It is also necessary to enhance the reliability of the GDO which is a fundamental data structure without which an object systems cannot continue to operate.

Time efficiency is a major concern. All references to objects typically require (possibly cached) access to GDO entries. All processing in an object system is performed as method invocations on objects and this means that GDO accesses must be highly efficient. Any GDO design must explicitly address the issue of fast access to the data within the GDO and any of its sub-structures.

Space efficiency in the storage of the GDO is also a primary concern due to the large number of objects, and hence GDO entries, that must eventually be supported in large scale distributed object systems. To achieve this goal, a GDO design must support the factoring out of common information from multiple GDO entries into their own structures. For example, all objects instantiated from a given object type and hence members of the same class will share common management information which is based on their class membership. To avoid redundancy and the corresponding space overhead and consistency concerns, this information should be removed to a separate sub-structure of the GDO which is for class-specific information. This information is then accessed from within GDO entries via references.

The wide range of possible failure modes in distributed systems is well known. To address the possibility of individual system failures affecting the operation of the GDO and the functions based on it, some mechanism for supporting enhanced reliability must be provided. Some form of replication together with a supporting consistency protocol is a reasonable alternative.

Finally, scalability is a must. Large persistent object systems, particularly those based on DSVM, may grow in size from a handful of machines to many thousands or more. For this to be possible, it is necessary for the GDO, a core system component, to scale with the rest of the system.

To further complicate matters, many of these design goals, as discussed, are mutually contradictory. For example, supporting replication normally implies increased overhead in terms of messages sent to ensure consistency. Further, such messaging normally limits scalability. This is only one of a number of such complex inter-relationships between the desired design goals.

3 GDO Description and Characteristics

The GDO for a distributed persistent object system must be discussed at two distinct levels. First, the object management functions that drive the composition of each entry in the GDO must be discussed. Second, the overall structure of the GDO itself must be considered.

3.1 Entries in the GDO

The purpose of the GDO is to serve as a repository for information related to object management and, perhaps, as a service provider for management functions based on that information. With this in mind, the following brief discussion of the contents of a GDO entry is presented.

The entries in the GDO must be distinguished from one another and explicitly associated with the object they describe. To accomplish, this, each GDO entry must contain the Object IDentifier (OID) of the object it describes. In the case of our DSVM system, the OID for an object is the virtual address at which it occurs in the persistent object space. The OID serves as the key field for lookups in the GDO. In a conventional persistent object system, the OID is provided as a part of an object reference. In our DSVM system, the OID is simply the virtual address of the object and *is* the object reference.

In addition to the OID much additional information also needs to be stored in a GDO entry. This information may be divided into three basic categories; object representation information, replication sensitive object management information, and replication insensitive object management information.

Some of the most obvious data to include in the GDO is object representation information. This includes references to persistent storage (i.e. disk) locations for the various components that make up an object's representation in the shared virtual memory including, at least, its methods and attributes.

Other fields in the GDO may include those which contain information which is sensitive to the replication of the GDO. By sensitive, we mean that replication of the information introduces problems which may not be practical to overcome. A good example of such data can be seen if we assume that concurrency control (an object management function) will be provided on a per-object basis (i.e. "object level locking"). In this case, an obvious field to have in each GDO entry is the actual lock for the object. This informa-

tion, however, is sensitive to distribution. If we implement reliability through replication, then we cannot freely replicate the lock field since a lock must be the *single* point of control for access to the corresponding object.

Other possible fields within a GDO entry may be insensitive to replication. For example, a field that describes class membership, may be safely duplicated in multiple GDO replicas.

Without considering a particular design, it is not possible to describe a GDO entry in detail. This is because some information contained in a GDO entry is specific to the distributed implementation of the GDO as a whole. Additional information concerning the contents of GDO entries will be provided later in the paper once the proposed design has been presented.

3.2 The GDO's Structure

The GDO as a whole is basically a search structure whose entries describe individual objects. The possible operations on objects include creation of new objects, (logical) deletion of existing objects[4], and invocation of methods on existing objects. These operations dictate two required operations on the GDO: *entry creation* and *entry lookup*.

A given object is only created once but may, if long-lived, be looked-up (to allow method invocations on it) very many times. Clearly the goal of any data structure implementing the GDO should consider this and provide fast searching for, and "reasonable" addition of, entries[5]. Additionally, the GDO may contain a very large number of entries (even if partitioned, as we will suggest later in this paper) and the number of entries may grow from relatively few to many. The data structure used to implement the GDO must support these characteristics.

To achieve the search efficiency required, either a multi-level index structure or some form of hashing is required[6]. Both of these alternatives provide the necessary characteristics. For a variety of low-level implementation related reasons, we chose to use a B+-tree. The B+-tree offers natural growth, fast searching, and acceptable creation overhead. It also is an easily subdivided structure and one which can be readily cached in memory.

[4] Objects are considered to be immutable so that their OIDs are never reused.

[5] In a distributed design, these guidelines also apply to messaging. Increased communications may be acceptable during object creation but not during object lookup.

[6] This is supported by the selection of these structures for use in existing object stores such as Exodus [CDG+90].

4 Evaluation of Potential Designs

An important issue to consider in designing an object directory is the placement of the directory in the system. While the object directory has, until this point, been considered a single entity managed by one node, it proves worthwhile to take advantage of the distributed nature of a DSVM system when placing the directory so that the directory itself becomes distributed. There are several ways to place the object directory in a distributed system. The directory can be centralized at a single node with no replication of the directory anywhere. At the other extreme, the directory can be fully replicated at every node. The directory can also be partitioned or fragmented and the fragments can be distributed among some or all of the nodes. Additionally, the directory can be fragmented and distributed among some nodes while other nodes maintain replicas of the fragments.

When deciding on a strategy for placing the object directory, in a DSVM system, it is important to consider how a particular strategy will affect the stated design goals. One such goal is to provide reliability in the system so that the failure of a node maintaining an object directory will not seriously impair access to the objects in the system. Replication is most often used to address reliability concerns but the need for reliability must be balanced with the desire to allow the consistency of the directory to be easily maintained. Consistency is difficult to maintain in the face of replication unless the replicas of a data item can be locked before modifications are performed. Establishing locks on replicated items can result in a significant amount of communication overhead and an important concern is to try and minimize communication overhead to prevent the network from becoming a bottleneck. An additional consideration is to reduce the amount of storage overhead needed to maintain the directory. Clearly, a single directory that manages a large number of objects can consume a great deal of storage space at a single site[7]. The size of the directory can also affect the ability of the directory to be searched quickly; this can in turn affect the latency time for servicing directory service requests.

Although in a DSVM system, the autonomy of individual nodes in matters such as concurrency control and the servicing of non-local requests for data items is fairly restricted, one form of autonomy – a node's autonomy in choosing a name or identifier for an object – can be influenced by the choice of placement strategy for the object directory. Garcia-Molina and Kogan describe two forms of naming autonomy [GMK88]. Name creation autonomy refers to a process' freedom in selecting any arbitrary identifier for a newly created object. Name registration autonomy

[7] The storage space required for all objects might easily exceed the storage capacity of a single node in the distributed system.

refers to a process' ability to register a systemwide object identifier without having to communicate with other nodes to determine the uniqueness of the identifier. Since all identifiers for persistent objects must be registered in an object directory, the placement of the directory can affect a node's name registration autonomy.

4.1 A Centralized Directory

By centralizing the object directory at a single node, that node is made responsible for maintaining the directory and its persistent storage on disk. Directory services such as the validation and registration of object identifiers, the location of attribute and method code segments in persistent storage, and the setting and clearing of locks, all have to execute at a single node. It is unreasonable to transfer the directory to each node invoking a service (so that each node can execute a service locally) since the overheads involved in transferring pages of the directory and ensuring their consistency is significant.

A centralized directory offers some advantages. The consistency of the directory is easy to maintain since there is only one copy of the directory. Also a single processor performing directory services essentially serializes the operations performed on the directory. As well, if the directory manages *relatively few* objects, the single directory can be searched fairly quickly without the need for communications with other nodes. This can result in reduced latency times for servicing requests for directory services.

The disadvantages of a centralized directory scheme, nonetheless, are significant. By maintaining the object directory at a single node, that node can potentially become the single point of failure for the entire system. Because the object directory provides the only mapping from virtual addresses to persistently stored segments, the loss of the object directory causes every persistently stored object to become inaccessible. Network traffic involving the node with the directory is high because every process faulting on an object or trying to obtain or release a lock on an object has to communicate with the node maintaining the directory. Li and Hudak [LH89] point out that if there are many object or page faults, the central node maintaining the locks will become a traffic bottleneck. This increases the latency time involved in servicing requests for directory services. Further, the storage requirements for the central directory will be large if the DSVM system manages many persistent objects. Finally, name registration autonomy is hindered in a system with a centralized directory since processes creating objects have to communicate with the node containing the directory to register their objects.

4.2 A Replicated Directory

The entire object directory can be replicated and stored at multiple nodes (or all nodes) in the system. With directory services performed at each node maintaining a directory replica, requests for directory services are distributed. A node requesting a directory service can route its request to either the nearest node with a directory replica or to a node with a replica that is not very busy. If the directory is replicated at all nodes, requests for directory services can, in fact, be serviced locally. With access to the object directory distributed in this manner, the likelihood of a node becoming a bottleneck is reduced. Also, by replicating the directory, the reliability of the system is enhanced. The failure of a single node maintaining a replica of the directory does not preclude accessibility to any object as long as a node requesting a directory service can route its request to another replica – no single node can become the single point of failure in the system.

Despite these advantages provided by the replication of the object directory, the drawbacks to the strategy make full replication impractical. Preserving the consistency of the directory is extremely costly. Information, such as the storage locations of segments, will probably be updated infrequently so modifications to that information can be propagated to other replicas in a lazy fashion. Locking and cache consistency information, though, will be updated far too frequently for the lazy propagation of updates to be practical. Instead, the pages of the directory could be locked before the directory is updated. However, the pages would have to be locked at every node with a replica of the directory. If the directory is replicated at all nodes, each modification of the directory would require a lock request to be broadcast to all nodes. This would result in an excessive amount of network traffic. Also, just as with a centralized directory, the storage requirements to maintain a directory replica will be excessive if the DSVM manages a large number of objects. In fact, the storage requirements are greater in a system with a replicated directory because storage space is consumed at multiple nodes. In addition, name registration autonomy does not significantly improve under a replicated directory system when compared to a centralized directory system. Although a process can register an identifier at a local node if the directory is fully replicated, the nodes must communicate with each other to either lock all the directory replicas or to achieve some sort of consensus on the new object identifier's uniqueness.

A common approach to dealing with the overhead of maintaining exact and immediate consistency between replicas is to use a master/slave scheme such as that commonly used in name services for existing Unix-based systems. Unfortunately, this approach is inadequate for the design of a GDO because stale information cannot be tolerated.

4.3 A Partitioned Directory

In this strategy, the object directory is partitioned (or "fragmented") and each fragment is placed at a distinct node in the system. Each fragment contains directory entries for some disjoint subset of the persistent objects in the system. Each of those nodes maintaining a directory fragment becomes the "home site" for the subset of objects listed in the fragment. The home site and its directory fragment then becomes responsible for providing the directory services for requests involving the objects covered by the fragment. In fact, partitioning the directory actually partitions the virtual address space so that each directory fragment covers some range(s) of virtual addresses. This results in a uniform distribution of directory service requests across all directory fragments if, as in our DSVM system, object identifiers are assigned randomly. To enable requests for directory services to be routed to the proper home sites, a map structure is needed at each node. This structure maps a virtual address range to the node with the directory fragment covering that range (the home site). The DASH multiprocessor [LLG+92] and the Ivy DSM system [LH89] are both examples of systems that partition the directory and address space and distribute fragments to all nodes. Unlike Munin [CBZ95], which allows directory entries for objects to migrate to nodes caching the object, DASH and Ivy directories reside at fixed home sites for each data object.

The advantages of partitioning the object directory are clear. Requests for directory services can be distributed to the correct home sites and this reduces the likelihood of a single bottleneck forming. Since the partitions are not replicated, it is easy to ensure the consistency of the directory fragments. If the directory is partitioned in such a way that there are as many fragments as there are nodes in the system, name registration autonomy is enhanced because each process creating an object can register its identifier at a local directory fragment. This constrains name creation autonomy, though, since a newly created object identifier has to fall within the range of addresses covered by the local directory fragment. An additional benefit provided by a partitioning strategy is that the storage requirements for maintaining the directory at individual nodes are quite small since each fragment will be smaller than the total directory. The total storage for the directory is effectively distributed among several nodes. As well, the latency for servicing directory requests would be lessened since the smaller directory fragments can be searched more quickly. Finally, the concept of a home site for each object fits well with the use of LRC for consistency control. Lazy Release Consistency (LRC) [KCZ92] requires a home site for each data object to maintain such information as the last updater of the object and the identifier of the node with exclusive access to the object. This is precisely the kind of information that can be found in the object directory. Since with LRC, any access to an object must involve communication with the home site before a lock can be granted, the use of a partitioned object directory would not incur any significant additional communication overhead.

Although a partitioned directory can provide a number of benefits, there are still drawbacks involved with the strategy. First, partitioning the directory without using replication does not improve the reliability of the system greatly since the failure of a node maintaining a directory fragment can cause some subset of the objects in the system to become inaccessible. As well, objects that are accessed frequently can cause bottlenecks at the nodes that are their home sites. If the directory is not partitioned in a way that allows every node to maintain a directory fragment, name registration autonomy is much the same as in a centralized directory system. Some processes still have to communicate with a remote node to register their newly created objects. Furthermore, if the partitioning and allocation of fragments to nodes is done without considering a node's potential usage of the fragment's address range, nodes with much activity could consume their address space relatively quickly; nodes without much activity could essentially waste virtual addresses.

4.4 A Partitioned Directory with Limited Replication

Like the partitioning strategy, this strategy involves partitioning the object directory and assigning each fragment to a distinct node. Each fragment, though, is replicated one or more times and each fragment replica is assigned to a separate node. This strategy is a reasonable compromise between partitioning the directory and replicating the directory since it combines the desirable characteristics of both those strategies. Each node maintaining a copy of a directory fragment becomes a home site for the objects covered by the fragment's address range so map structures are still necessary at every node to map virtual address ranges to home sites. Replication of the directory fragments, though, allows requests for directory services involving a particular directory fragment to be distributed among the nodes with replicas of the fragment. As well, replication of the directory fragments enhances the reliability of the system. Even if a node with a directory fragment fails, another node with a replica of the fragment can still continue as a home site for the subset of objects covered by the directory fragment.

Despite the fact that a scheme involving a partitioned directory with a limited number of replicas possesses some of the advantages of both the partitioning and replication strategies, this hybrid strategy also possesses some of their

disadvantages. Because the directory fragments are replicated, the consistency of the directory can be difficult to maintain. Although there are only a limited number of replicas of each fragment, and this makes the propagation of updates involving a directory fragment easier than in a system with gross replication of a directory, the lock variables, sharing lists, and lock request queues in each directory fragment must be updated far too frequently for the lazy propagation of updates to be sensible. Locking directory pages in each replica, however, results in less network traffic in a system with limited replicas than in a system with gross replication of a directory since there are fewer copies to lock. An additional drawback to this hybrid strategy is that the name registration autonomy of individual processes retains the problems of the partitioning and replication strategies. The registration of a new object identifier involves invoking a directory service on some remote node maintaining a directory fragment. As well, the registration of an object identifier still requires the nodes with replicas of a fragment to communicate with each other to achieve consensus regarding the identifier's uniqueness.

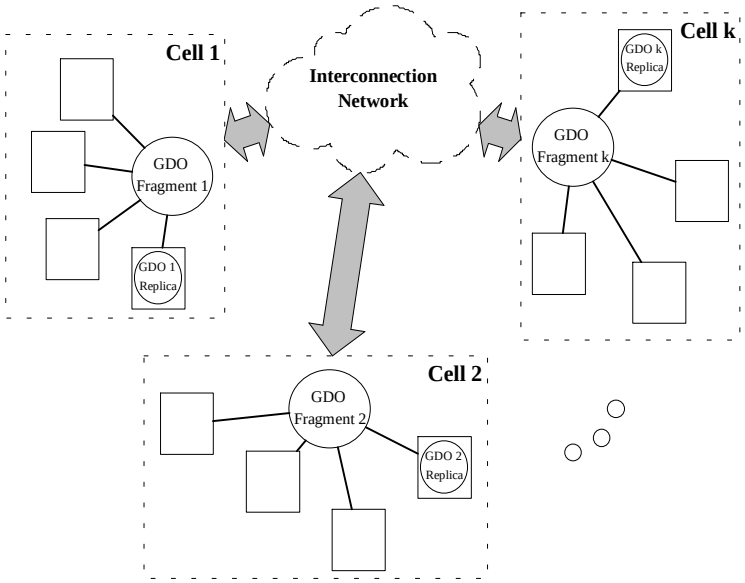

Figure 3: Placement Strategy for GDO Fragments

5 The GDO Design

Some of the issues surrounding the contents and placement of an object directory in a DSVM system have now been examined and a reasonable high-level design of an object directory can be presented. The selected implementation data structure for the GDO is a B+-tree. This is appropriate since a B+-tree provides fairly compact storage of the directory while providing for fast searching. Graham, et. al [GBBZ93] discuss the relative merits of using a B+-tree to store object directory information and point out that a B+-tree may be effectively cached to improve performance and that it supports key *range* searching[8]. The DSVM system design is assumed to consist of some number of nodes, each with local disk storage, connected by a high bandwidth, low latency network that can (directly or indirectly) support broadcasting.

5.1 Placement Strategy for the GDO

Of the directory placement strategies discussed in this paper, the scheme using a partitioned directory with limited replication of directory fragments provides the most advantages. For this reason, the GDO (and consequently the virtual address space) is partitioned into a number of directory fragments. The number of fragments should be one-half the number of nodes in the system or less to allow each fragment to be replicated at least at one node. The fragments should cover disjoint virtual address ranges of equal size and together the fragments must cover the entire virtual address space. Following the partitioning of the GDO, the nodes themselves will be clustered into cells of two or more nodes each. As expected, the number of cells equals the number of GDO fragments and one node in each segment will be allocated a GDO fragment. At least one other node in each cell should be allocated a replica of the cell's GDO fragment. Figure 3 illustrates such a placement strategy. It should be noted that the clustering of the nodes into cells does not necessarily reflect the topology of the underlying network but instead indicates the relationships between the nodes in a cell with the nodes maintaining GDO fragments. The clustering of the nodes does not impair the ability of any arbitrary node to either broadcast a message or communicate with any other arbitrary node. That being said, however, it would be sensible to cluster nodes together that can communicate with each other at low communication costs.

Name registration autonomy can be enhanced to some degree by allowing a process executing on some node to register a *random* object identifier at the GDO fragment within the node's own cell. While this does not provide complete name registration autonomy as registration at a local node would provide, this registration strategy only requires communication between the nodes in a single cell during registration. Name creation autonomy, however, is constrained by the registration strategy since a process creating an object is only allowed to create a random identifier within the address range covered by the cell's GDO frag-

[8] Key range searching of the GDO is required if method invocations are made to absolute virtual addresses rather than through the start of an object.

ment. Also, because a GDO fragment's address range is smaller than the entire address space of 2^{64} bytes, the likelihood of a randomly created object identifier colliding with an existing object's address range is likely to be higher with this registration method. To address this problem and the problem of constrained name creation autonomy, a process creating an object could also be allowed to create a completely random 64-bit object identifier. The process would then have to communicate with the remote node maintaining the GDO fragment covering the random address to validate and register the new object's identifier in the GDO fragment. This latter strategy has been adopted.

With each GDO fragment being replicated at one or more nodes, ensuring the consistency of the replicas can pose difficulties. Since changes to the attribute and code segment references should occur infrequently, a node with a GDO fragment can propagate those updates to the other replicas in a lazy fashion. The problem with maintaining consistency, instead, centers around the locking and cache consistency information. Lock variables, sharing lists, lock request queues, and last updater fields associated with LRC can be updated very frequently. To eliminate the problems involved in locking directory pages or continually transmitting updates, a GDO entry's locking and cache consistency information can be allowed to be inconsistent in all but one of the copies of a GDO fragment. Each copy of a GDO fragment then maintains locking and cache consistency information for only a *disjoint* subset of the objects covered by the fragment. The remaining information associated with a GDO fragment's entry, such as attribute segment and code segment references, can be easily kept consistent in all replicas of the fragment. Any node wishing to lock an object or release a lock on an object, for example, would then have to invoke the appropriate directory service on the correct node holding a fragment replica which handles the locking and cache consistency for the object.

To assist a node in locating the GDO fragment replica handling the locking and cache consistency information for a desired object, each node should persistently store a map structure that maps virtual address ranges to the nodes with GDO fragment replicas. The map structure should contain the base addresses of the virtual address ranges covered by every cell's GDO fragment. Each base address can then be mapped to a list of entries. Each entry will contain the identifier of a node maintaining a replica of the fragment along with the address range that the replica handles locking and cache consistency for. A node wishing to invoke some directory service for a particular object can first search the map structure for the cell containing the GDO fragment covering the virtual address of the object. The node can then traverse the list associated with the cell to find the specific node with the GDO fragment replica responsible for the object's virtual address. Requests for directory services involving that object can then be forwarded to the node with the appropriate fragment replica. The map structure only needs to be modified when the virtual address space is re-partitioned (when cells are added to or removed from the system), or if GDO fragment replicas were to migrate. Since these events will occur infrequently, the map structure can be implemented as a read-only structure. The system can thus scale with little cost as long as nodes are added to or removed from existing cells and those nodes do not maintain GDO fragment replicas.

With each GDO fragment having one or more replicas within its cell, the reliability of the GDO is enhanced. When a node maintaining a copy of the GDO fragment fails, another node with a copy of the fragment can take over for the failed node and perform directory services for the objects formerly managed by the failed node as well as for its own objects. Since each node has a map structure that lists the nodes with copies of a cell's GDO fragment, a client node that had invoked a directory service at a failed node and received no response can simply invoke the service at another, pre-specified, node in the map structure list. The other node will be the one that has taken over management of the GDO fragment replica belonging to the failed node.

While the node taking over from the failed node has directory entries for each of the objects covered by the cell's address range, it does not, at the time of failure, have consistent locking and cache consistency related information for the objects previously handled by the failed node. Before performing a directory service for a particular object not managed by the node previously, the node must obtain up-to-date information about which nodes currently cache the object and which nodes, if any, currently hold locks on the object or have pending lock requests. To obtain this information, the node can broadcast requests for this information the first time it is asked to perform a directory service for an object it did not manage previously. As long as each node in the system is aware of the fact that it either holds a lock on an object, has requested a lock on an object and the request is still pending, or is caching an object, the node broadcasting requests for that information can be assured of receiving up-to-date information. The page table at each node can maintain information such as whether or not the pages of an object are locked by the node and whether or not the node currently caches the pages of an object. To maintain information about pending lock requests, each node needs to implement some sort of queue structure to list the lock requests that originated at the node but which have not yet been serviced.

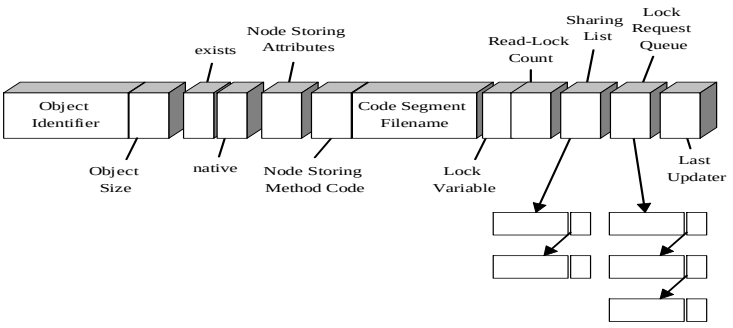

Figure 4: GDO Entry Format

5.2 Contents of a GDO Entry

Figure 4 illustrates a possible format for a GDO entry. This is only one of many possible formats and includes only a subset of the object management information that would be needed in a complete system. As an example, it is meant to be illustrative rather than exhaustive. Many of the fields were discussed in Section 3 and so are fairly straightforward. A GDO entry contains the base address of an object's attribute values to represent the object's identifier. This value is used as the key value for searching the GDO. As well, each entry also possesses a field indicating the object's size in bytes. The "exists" field indicates whether or not the corresponding object has been deleted.

Locking and cache consistency information is also maintained in the GDO entry. Each entry contains a field for the lock variable as well as a field indicating the number of read-locks currently held on the object. The entry also contains a pointer to a sharing list indicating the nodes that currently cache the object. Using an invalidation protocol, the directory, upon write-locking the object, traverses the object's sharing list and sends invalidation messages for the pages of the object to each of the nodes listed in the sharing list. Additionally, each directory entry contains a pointer to a queue for pending lock requests and a field that identifies the last node to write-lock the object (the last updater). A field that has not been discussed previously is the "native" field. This field indicates whether or not the particular GDO fragment replica maintains the locking and cache consistency information for the object. When a node with a GDO fragment replica receives a request to perform a directory service involving an object whose directory entry indicates that it is not "native" to the fragment replica, it indicates that the native fragment replica for the object has failed. To service the request, the replica taking over from the failed node must first broadcast requests for the locking and cache consistency information involving the object before performing the directory service.

The persistent storage location references for the attribute values segment and the method code segment both refer to index structures located on the nodes that persistently store the segments. Graham and Barker [GB93] suggest using a file structure to index a segment (or segments). The file structure for an object's attribute values can then be referred to using a filename consisting of the object's identifier. For this reason, the storage location reference for an attribute values segment consists only of the identifier of the node persistently storing the segment. A node faulting on an object can use the object's identifier to request the pages of the object's attribute values from the server node identified by the GDO. The storage location reference for a method code segment can consist of the identifier of the node storing the segment, along with a filename representing the virtual address of the class definition object that defines the methods.

To support flexibility, the structure of a GDO entry is variable. After certain required fields, OID, pointers to persistent storage, etc. all other fields are optional. This reflects the desire to be able to manage different objects differently. Since not all objects will require all management functions, there need only be fields corresponding to those functions specified. The varying length of each GDO entry is easily supported by the use of a B+-tree as the implementation data structure since the tree itself is built of index nodes containing fixed size keys (OIDs) and pointers. The actual data (entries in the GDO are stored in a separate data area and therefore need not be of a conveniently fixed size.

Additionally, while any two given objects may both require some management function, say concurrency control, the implementation of that function may differ between the objects. Thus, each GDO entry contains pointers to implementation specific data structures for each applicable management function rather than the information in the data structures itself. The interpretation of the GDO entry format and the data associated with each management function is based on a required, fixed format field at the front of each GDO entry which specifies the relevant object management information. This mechanism is too complex to discuss in detail but will be the subject of a forthcoming paper.

6 Design Evaluation

In this section, we evaluate how well the presented GDO design meets the goals discussed in Section 2.2.

The first stated goal was flexibility. To support flexibility, the design makes few or no assumptions about the structure of GDO entries beyond the requirement for a key

field and the expectation that additional fields will exist. As described, many of these additional fields are optional and may be interpreted differently from object to object.

The B+-tree structure is easily distributed. Not only can it be effectively partitioned as described in the paper but it is also amenable to in-memory caching which offers clear performance benefits. In particular, those tree nodes near the root node are all relatively static. They only change infrequently when index node splitting propogates to the top of the tree. Being nearly read-only, such nodes may be freely cached without fear of consistency or reliability woes.

Time efficiency is also provided by our design. The use of random OIDs ensures that newly created objects will be evenly distributed across all GDO fragments thus balancing the load of providing object management services. Further, the limited replication provides alternate points of service for all but the replication sensitive services. Although not discussed, the overhead of providing object management services will also be split between the node invoking the object method which requires management and the GDO fragment server which offers the required services. Wherever possible, the cost of object management services is borne by the "invoker" of those services. Finally, the chief limit on the effectiveness of many large scale distributed systems has been the need to exchange information between many sites. Conventional shared bus networks have not provided the needed bandwidth and low latency. With the development of ATM (Asynchronous Transfer Mode [D.E95]) and similar switched networking systems, this problem should be ameliorated.

The space efficiency of the GDO is also addressed by the use of a B+-tree as its implementation structure. In addition to being easily partitionable, which avoids needless replication, the B+-tree is naturally storage efficient when compared to many other multi-level index structures. Further, our choice to perform limited replication, when done intelligently considering enterprise induced topological network structure, not only enhances reliability and autonomy but also does so without incurring undue storage overhead.

The primary mechanism for reliability in the GDO is the use of replicas. We believe that limited replication can provide a high degree of reliability which will be sufficient to ensure the effective "non-stop" operation of the GDO. As detailed in the paper, an election scheme is required to deal with the failure of a node hosting a GDO fragment containing replication sensitive information. Fully distributed election algorithms are well documented and understood and it is a relatively simple matter to have each "client" system retain sufficient state information to permit an elected server replacement to rebuild the needed tables to permit continuing operation. As we believe that these failures will be relatively infrequent and since only a subset of objects will be affected, the comparatively long time required to perform and election and recover state is acceptable.

The final goal of our GDO design was to offer scalability. The partitioned nature of the design is one step towards ensuring scalability. We further argue that as the size of a system grows (in terms of the number of persistent objects it contains), so too will the number of machines in the distributed system. Because of the load balancing features that are inherent in our design, the addition of more machines that can serve as GDO fragment servers, means that the system should scale well. This is a distinct advantage of a fully distributed, "peer to peer" system rather than a conventional "client-server" system.

For the reasons just stated, we believe the GDO design presented in this paper meets all the stated design goals to the greatest degree possible. Further, the design is a practical and implementable one[9].

7 Related Work

A number of objectbases and distributed object systems have been developed recently with differing approaches to implementing directory structures. Amoeba [MvT+90] is a distributed operating system that supports the persistent storage of data objects. Amoeba's directory service, though, only provides a mapping from user-defined object names to capabilities defining access rights for objects. Once a process obtains a capability for an object, it must locate the server storing the object by broadcasting or multicasting a request for the capability. The server storing the object can then transfer the object to the client node the process is executing on. The ObjectStore system [LLOW91] also supports persistent objects and ObjectStore directly controls the virtual memory used by the application. However, directories involving virtual addresses, such as the virtual address map and tag table, are not persistently stored and must be recreated each time ObjectStore starts up. As well, ObjectStore does not support distributed clients and servers.

Software supporting distributed heterogenous objects such as Corba [OMG93] must also provide basic object location services. Similar facilities are also required under ISO's GDMO proposal [ISO92]. In the context of such work, the GDO described in this paper might be considered a highly advanced ORB providing many other services in addition to object location.

Distributed shared memory systems use directory structures similar to cache coherence directories in shared-memory multiprocessors. Like the DASH multiprocessor [LLG+92],

[9] An initial implementation of the system is currently underway and should be completed by the summer of 1996.

both Munin [CBZ95] and Ivy [LH89] maintain directories at each node. Since Munin does not support persistent objects, a Munin object directory only contains cache coherence information. Ivy can support persistent objects and so its directory can be used to locate where an object is likely to be stored. Ivy, though, uses several distinct directories for providing locks and consistency control.

The Opal system [CLFL94] implements a SVM for conventional, non-object oriented, programming. A directory structure within Opal maps virtual addresses to segment identifiers. Using another directory structure, segment identifiers are mapped to the backing files that persistently store the objects. Access rights to segments and consistency related information are stored in separate directory structures. Opal is not yet fully distributed and does not provide automatic persistence.

The DSVM system proposed by Tam and Hsu [TH90] associates a directory structure with each node in the system. This directory, called a locator, maps the address range of a virtual memory page to the node that is currently updating the page – the page's owner. Each node maintains a directory for the pages it owns and that directory associates a token and a list of nodes sharing the page with each owned page. While the ownership of a page can migrate between nodes, each page is always associated with a specific locator node. Again, separate data structures are used to indicate the storage locations of objects, access rights to the objects, and consistency control information for the objects.

8 Conclusions and Future Work

In this paper we have described the design of a Global Directory of Objects (GDO) for a distributed persistent object system that maintains object management information and supports object management functions for all objects. The design provides both time and space efficiency, reliability, flexibility, and scalability in a fully distributed system. We believe that the design is practical and are currently implementing a prototype system to prove it. Although the work presented here is for a somewhat unique object environment, much of it is generally applicable to more conventional distributed object systems.

In addition to the proof of concept implementation, there are numerous open questions which will be the subject of future work. Among these are strategies for the placement of the limited replicas given knowledge of the network structure and organizational boundaries, the design of efficient protocols for delivering the needed object management services, and, in particular, consideration of the problem of specifying and maintaining the object management information for very large persistent object systems. In this final area, we are studying a concept we refer to as Management Nets (or M-Nets) as a mechanism for managing the management information.

Lastly, there is a noteworthy similarity between the information contained collectively in the GDO and the related M-Nets and meta-information as discussed by object system theoreticians. Eventually, we hope to make the entire DSVM system a uniform reflective environment in which case the GDO and M-Nets will be subsumed as first class objects themselves.

References

[BPG95] K. Barker, R. Peters, and P. Graham. Distributed Shared Virtual Memory (DSVM) for Interoperability of Heterogenous Information Systems. In *OOPSLA Workshop on Object Interoperability*, 1995.

[CBZ95] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques For Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[CDG+90] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenburg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufmann, 1990.

[CLFL94] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[D.E95] D.E. McDysan and D.L. Spohn. *ATM : Theory and Application*. McGraw Hill, 1995.

[GB93] P. Graham and K. Barker. Distributed Object Base Implementation Using a Single Shared Address Space. In *Proceedings of the Mid-Continent Information Systems Conference*, pages 62 – 77, 1993.

[GBBZ93] P.C.J. Graham, K.E. Barker, S. Bhar, and M. Zapp. A Paged Distributed Shared Virtual Memory System Supporting Persistent Objects and Engineering. Technical report, Department of Computer Science, University of Manitoba, 1993. *unpublished paper - available on request*.

[GBPZ95] P. Graham, K.E. Barker, R. Peters, and M. Zapp. Distributed Shared Virtual Memory for the Distributed Management of Persistent Objects. Technical Report TR-95/12, Department of Computer Science, University of Manitoba, 1995.

[GMK88] H. Garcia-Molina and B. Kogan. Node Autonomy In Distributed Systems. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 158 – 166, 1988.

[ISO92] ISO. *Information Technology – Open Ssystems Interconnection – Structure of Management Information – Guidelines for the Definition of Managed Objects: ISO/IEC 10165-4:1992*. International Standards Organization, 1992.

[KCZ92] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth Symposium on Computer Architecture*, pages 13 – 21, 1992.

[KK93] D. Kemper and A. Kossmann. Adaptable pointer swizzling strategies in object bases. In *Proceedings of the* 9^{th} *International Conference on Data Engineering*, Vienna, April 1993.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–70, March 1992.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[Mos85] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[MvT+90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[OMG93] OMG. *The Common Object Request Broker: Architecture and Specification – Revision 1.2.* The Object Management Group, Dec 1993.

[TH90] V. Tam and M. Hsu. Fast Recovery in Distributed Shared Virtual Memory Systems. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pages 38 – 45, 1990.

[WD92] Seth J. White and David J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the* 18^{th} *International Conference on Very Large Data Bases*, Vancouver, British Columbia, October 1992.

[Wei89] W.E. Weihl. Theory of Nested Transactions. In S. Mullender, editor, *Distributed Systems*, chapter 12, pages 237 – 262. ACM Press, 1989.

[WK92] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS92)*, pages 364–377, 1992.