# Distributed object memory schemes and applications: A literature survey

Jeff McAffer

October 7, 1994

## 1 Introduction

Previously, computational and communicative irregularity have been identified as sources of difficulty in building concurrent systems. We now add another dimension of object fragmentation. We propose that the degree of fragmentation will be a differentiating factor for applications implemented in multi-space systems. This has been indirectly suggested in Shapiro's work on Fragmented Objects [Shapiro et al., 1989] but does not appear to have been directly treated in the current literature.

With the advent and spread of distributed systems has come a number of schemes for handling object distribution. Unfortunately, we are not convinced by the applications used to test and "prove" these schemes. While we do agree that they represent reasonable uses of distributed objects, they are, by in large, algorithms rather than applications. That is, they are distributed systems applied to some problem rather than a solution to some distributed problem.

We are interested in situations where distribution is forced rather than just desirable. Having said that, we actually think in terms of *multi-space* systems rather than distributed systems. Physically distributed systems are of course or great interest but there are cases where programs running on the same physical machine cannot directly communicate and share data. For example, cooperating C++, Smalltalk and CLOS objects. For all intents and purposes, these objects are distributed across a network of machines with very low latency.

With that brief introduction, we will proceed to describe the findings of our literature survey. Please note that we have pruned only the most obviously unrelated papers from the report below. Papers thought to be marginally relevant now may become directly relevant as our work progresses.

## 2 Search

We looked primarily for work in two different areas; implementation techniques and mechanisms, and example applications of object-oriented multi-space systems. We cast a wide net and covered areas including; distributed shared memory (DSM), memory caching/paging, database caching, persistent object swapping and pointer swizzling, object/data replication and migration, automatic garbage collection, distributed operating systems, distributed object-oriented systems, language support for distributed computing and applications of distributed symbolic computing.

We do not claim to have encountered every paper of import in all of these areas but propose that the work investigated is representative of these areas as they apply to the goals of our project. The papers surveyed are detailed in a companion document, "The Red list".

Overall there was not much in the current literature which directly compares with our work. There are many schemes for implementing various forms of multi-space systems. Some are quite close to the our testbed system and we will be able a valuable source of techniques. Mechanisms are only half the story however. The database and scientific or algorithmic areas contained many example applications. Unfortunately, most had one or more of the following drawbacks (with respect to our domain):

1. Limited number of object spaces. Many systems use broadcast techniques in various ways (e.g., maintaining replica consistency). While the cost of this is often more a function of object particulars (e.g., the number of replicas), for systems with widespread object interdependency, inter-space dependency is unpredictable.

Other systems focused mainly on database applications which can roughly be classed as two-space systems — in the persistent store or in transient storage.

2. Regularity. Many applications are drawn from the scientific parallel computing world (e.g., successive over relaxation, N-body). While some have inherent some degree of computational or communicative irregularity, few have large amounts of object fragmentation or interrelation irregularity.

3. Limited object fragmentation. Related to the above, few applications had more than say, five, different kinds (classes) of objects or interconnection behaviours. While this is not an absolute prerequisite of fragmentation irregularity, it seems to be a good indicator.

This is not to say that there is nothing of use but rather that there is nothing which directly compares to our work. The findings of our survey are detailed in the sections below.

# 3 Mechanisms and models

The overall concept of an object does not change in a distributed system — They are still encapsulations of data and operations on that data. There are however, a number of implementation variations which can influence the capabilities and performance of an object system. The most straightforward object model uses some notion of proxy [Shapiro et al., 1989] or remote reference object as a local representative of a remote object. Messages to proxy (really to the remote object) are packaged up and forwarded to the remote site for processing. While functional, this is roughly equivalent to standard RPC techniques and can prove costly in particular situations.

To address these issues, various schemes for representation, replication and migration have been proposed. Perhaps the proposal which departs the most from typical object models is Fragmented Objects which appear in SOS [Shapiro et al., 1989] and [Dickman and Makpangou, 1992]. The key points of the Fragmented Object model are given below in the discussion of SOS. It could be said that this technique reduces to so-called smart proxies but in fact the proxies *are* the object.

In related areas, we found a great deal of prior art in hardware and software supported page-based schemes for DSM. In general this work, though well-developed, is not applicable to dynamic object-oriented systems. Several researchers [Hosking and Moss, 1993, Bal and Kaashoek, 1993, Tanenbaum et al., 1993] noted that object-based software schemes have more natural control semantics and can outperform the page-based hardware schemes.

There is considerable overlap between our work and the outlined object-based multi-space schemes. Several papers discussed the merits of separating application code from object/memory distribution mechanisms. Some detailed meta-architectures of varying levels of sophistication and flexibility (see the section below on meta-level support). These systems form the pool of techniques from which we will draw test environments for our experiments. We are confident that the current CodA meta-architecture is sufficiently powerful to implement any of the mechanisms in a way which does not impact the test application (base-level) code.

Work in the database area focused on two major areas; data loading and replication. Data loading is closely related to DSM as it involves issues of pointer swizzling, paging, sharing, latency, caching, etc. Replication has been shown to be of considerable benefit where data use and access costs are non-uniform [Bal and Kaashoek, 1993, Bennett et al., 1990]. As such, we find it of some interest.

Below we have grouped various distributed object models and mechanisms very roughly by the key feature or aspect they treat.

## 3.1 General systems/languages

Here we give broad outlines of some of the systems and languages we reviewed. Details of specific points/areas of interest are given in subsequent sections.

**Amadeus** [Cahill et al., 1993a] is an implementation of the Commandos [Cahill et al., 1993b] architecture and is a general purpose object-oriented programming environment. It contains a *generic runtime* (GRT) on which object-oriented language programmers can design and implement their object models. In essence it is a language builder's toolkit. The language need not be entirely new. For example, an extended version of C++ has been built on top of the Amadeus GRT.

The system operates at a relatively low-level (e.g., object headers etc.) and provides several useful capabilities. The runtime essentially forms (part of) the meta-level of the implemented language. One interesting feature is *clustering*. A cluster is a managed group of objects. Newly created objects are allocated in the newest cluster unless some other cluster is explicitly specified. Though it is not discussed in great detail, the authors hint that the GRT incorporates a heuristic for determining when new clusters should be created. We are quite interested in the effectiveness of this mechanism.

**Amber** [Chase et al., 1989] is a parallel programming system designed to run single applications efficiently on networks of multiprocessors. It follows in the footsteps of other sytsems taking its thread model from Presto [Bershad et al., 1988] and its migration and location operations from Emerald [Jul et al., 1988]. Amber provides a flat name space for fine-grained objects which can migrate between network nodes. It is assumed that each node is a shared memory multiprocessor (e.g., Firefly). Object level coherence is maintained by migrating threads rather than objects. That is, in normal execution, threads partially migrate to the node containing the object through which they wish to run.

Programmers are given, and expected to use, complete control of the details of object and thread location. Since object location dictates where threads will execute, objects can be explicitly and dynamically placed (migrated). Any executing threads will move with the object. Object replication is supported but only for immutable objects.

**Amoeba** [Tanenbaum et al., 1990] has an OS-like architecture and is mainly designed as infrastructure for interconnecting multiple computers. It's kernel and associated servers provide all the features that one would expect (e.g., RPC, threads, naming). Amoeba is the base for Orca [Bal et al., 1992, Tanenbaum et al., 1993] which defines and provides the higher level functions that one might expect/require in a distributed environment. On interesting feature of Amoeba is its *capabilities*. Every object has an associated capability which is like a key which succinctly describes the object, what it can do, where it is from and what privileges it has. Capabilities are encrypted and verified during RPCs and so are used to maintain security. This is an interesting system and is representative of the state of distributed system infrastructure but is not particularly applicable here as we are more interested in the higher level issues raise in say, Orca.

**Argus** [Liskov, 1988] is a language and system for running long-lived information systems (e.g., inventory control systems or mail systems). As such it must maintain integrity in the face of failures. It supports object migration and replication. Argus, the language, also contains constructs for *guardians* and *actions*. Guardians are essentially protected groups of objects accessible only through particular handlers. These are essentially multi-threaded actors where the handlers are the public interface. Actions are serializable, atomic sequences of operations; that is, transactions. The work on Argus does not seem to consider object migration and replication policies rather than mechanisms. Regardless, this work is interesting because they have applied it to some real-world problems (e.g., banking system) albeit from a transactional and consistency point of view.

**COOL** [Chandra et al., 1994b] is an object-based parallel programming language. Much of the work on COOL has gone towards synchronization issues but it does also address object distribution as it relates to effeciency. The main notion in this regard is *affinities*. Programmers can specify various affinity hints in parallel functions. For example, they can state that the function is best run local to some object or some processor. Task affinity describes parallel tasks which should run back-to-back. Presumably to exploit cache locality. COOL also allows for object migration through the use of explict migration functions. It does not appear to support replication.

These hints would appear to be powerful tools but they do require the programmer to know quite a bit about the underlying machine architecture. Fortunately, they do not affect the semantics and so can be changed with little difficulty.

**CORBA** *where did I put that document?* xxx

**NeXTStep** xxx

**Emerald** [Jul et al., 1988] is an object-oriented language and system designed for the construction of distributed programs. It supports fine-grained objects and migration. Emerald objects exist in a global uniform object space and cannot be replicated. A number of parameter passing schemes such as call-by-visit, call-by-move, call-by-value and call-by-reference are also supported.

The designers of Emerald opted to build a new language rather than adapt of modify an existing language. This is justified by the language and compiler they desired. Based on these attributes, the compiler can determine the appropriate (with respect to distribution) representation and behaviour for objects. It would perhaps be better to have defined a meta-architecture which could dynamically incorporate compiler information and determine the most effective representation and behaviour. More recent work [Jul, 1993] has headed in this direction but only to the extend that the concerns of objects and distribution are separate.

All in all, Emerald has a number of interesting features which are of interest to us in the design and implementation of our testbed environment.

**Midway** [Bershad et al., 1993] is a distributed shared memory programming system. It supports multiple consistency models concurrently and introduces a new model; *entry consistency*. Entry consistency recognizes that many programs place explicit synchronization locks around code regions which access shared data. Following this, whenever a lock is requested on an object, the object is moved to, and becomes owned by, the requesting processor. Updates to the object are then carried on an local basis and need not be broadcast. A particular processors view of the shared memory it will access is made consistent when it enters this lock region. Midway also supports release and processor memory consistency.

**Munin** [Bennett et al., 1990] supports distributed shared memory using a type-based approach to coherency. The goal is to support the execution of shared memory parallel programs on distributed memory machines. They identify, based on usage, nine different kinds of objects in typical shared memory programs. They are: write-once, private, write-many, result, synchronization, migratory, producer-consumer, read-mostly and general read-write. Using information gathered by the compiler and supplied by the user they identify particular objects in a program as being one of the above types. This in turn is used to determine the memory coherency scheme used to support that object. For example, read-mostly objects would perform well under a replication scheme.

This is quite an interesting system. The most recent paper we reviewed was 1990 and they did not seem to indicate any particular test or results which illustrated the benefit of these techniques. We do not doubt that they would be of use but are curious to know how, when and where.

**Orca** [Bal et al., 1992, Tanenbaum et al., 1993] is an object-based language for parallel programming on distributed systems. It is based on the Amoeba operating system [Tanenbaum et al., 1990]. Orca objects are passive and relatively coarse-grained. Inter-object communication and cooperation is based on the idea of *shared objects* which are sequentially consistent with respect to replica updates but do not enforce any update ordering.

Object replication is system-wide or not at all and is supported using Amoeba's totally-ordered broadcast mechanism. The Orca compiler gathers object read/write statistics and creates hints which are then used by the runtime system when determining object replication behaviour [Bal and Kaashoek, 1993]. For example, objects which are read often but written infrequently can benefit from replication.

A recent paper [Tanenbaum et al., 1994] demonstrated Orca's use in several applications (e.g., computer chess and neural net training). These examples, while compelling are not representative of systems ubiquitously constructed of objects. Nonetheless, Orca is interesting in its use of compile-time information at runtime and as a concrete example of efficient system-wide replication in certain situations.

**SOS** [Shapiro et al., 1989] is an object-oriented OS with typed objects which support distributed migration. Objects, Elementary Objects, are based on the C++ model and, due to Id management overhead, are medium grained. Elementary objects are grouped together as *fragments* of *Fragmented Objects* (FOs). The fragments of a FO can live in different object spaces. Each fragment represents and is equivalent to the whole. The actual internal fragmentation structure of a FO is transparent to the user who can see only the public interface. SOS also supports a wide-range of now common distributed object mechanisms (e.g., proxies, references). Migration plays a key role in the system. It is the base of FO growth and the building of cross context links. FO fragmentation structure and group membership is preserved by migration.

Fragmented Objects appear to be a powerful abstraction for structuring and managing distributed objects. As is pointed out in the work on VDOM [Feeley and Levy, 1992], FOs can be used to handle the implementation of algorithms which naturally partition computations. For example, many scientific applications contain grid-based or stencilling algorithms (e.g., Successive Over Relaxation). In such systems there is a logical object (e.g, rectangular grid to be relaxed) and computations on particular grid cells have well-defined interdependencies. FOs can be used to build a "grid" object out of "cell" fragments, each of which understands and copes with its dependencies.

While Fragmented Objects and other concepts in SOS are interesting and useful, the system is somewhat confused by the lack of a distinct meta-level. The details and mechanisms of handling inter-fragment communication and cooperation are grafted on to "normal" object behaviour in a special case way. Certain objects in the system have special privileges which allow them to directly access or invoke other objects. The base and meta-levels appear to be mixed and not enough of the underlying mechanism is hidden from the user/programmer.

## 3.2 Replication and caching and coherence

Schemes for replication vary widely. Nitzberg, [Nitzberg and Lo, 1991], gives a good review of various Distributed Shared Memory (DSM) techniques with references and definitions.

Bal and Tanenbaum have recently focused development of Orca on techniques for determining whether a particular object should be replicated [Bal and Kaashoek, 1993, Tanenbaum et al., 1993]. In Orca, replication is done on an everywhere or nowhere basis and is done using Amoeba's reliable ordered broadcast. The replication policy is determined by looking at compiler-generated information relating to shared-object read/write information. Each Orca "process" (roughly a concurrent procedure) is annotated with its variable read/write pattern. When a process is about to run, a check is done to see if the data it manipulates should be replicated. This is done by looking at who else will access it and the ratio of reads to writes. They present some good results in applications such as travelling salesman and shortest path.

Munin [Bennett et al., 1990] is similar to Orca in that they utilize compile-time object access information in the decision on how to manage object replication and coherence. Where as Orca includes one replication policy and one mechanism, Munin has some nine different policies which are based on various mechanism. For details see the section above on Munin.

VDOM (Versioned Distributed Object Model) [Feeley and Levy, 1992] has a great deal in common with Fragmented Objects [Shapiro et al., 1989] but differs in its model of replica consistency or coherence. VDOM introduces the notion of replica versioning. All objects are immutable. As a result, modifying an object creates a new version. When an operation is to be done on an object a specific version is selected and used throughout. This forms the basis of a weak consistency model.

Caching is a form of replication which is often used in systems having a large disparity in access latency or bandwidth (e.g., persistent stores). Burdorf [Burdorf, 1993] has done a study of various policies for caching persistent objects in transient memory. Several caching schemes (trace-based, LRU, FIFO etc) are evaluated in several different applications (e.g., TSP, shortest path).

There are no hard conclusions but they do note that in some situations, given a limited cache size, it is best to dispose of the most frequently read element. This somewhat counter-intuitive result is based on the idea that some computations are bursty — They use a set of objects and then move on. The more frequently accessed objects are less likely to be used in the future because they will have already been processed. Of course, some sort of delay must be imposed to avoid tossing pages still in use. Still, this result is interesting because it shows that the nature of an application's computation can and should play a large role in determining the strategies/policies used.

Work in Distributed Caching [Pu et al., 1992] is aimed at systems which naturally contain data servers though we feel it may have a wider appeal. . The basic idea is similar to compute load balancing but considers data access as the scarce resource. Access costly objects are cached in the transient memory of *mutual-servers* rather than the host which actually holds the object. By using broadcast messages for object access, the fastest (least loaded) node in the system which has a replica will answer.

This of course has some drawbacks. Assume that the fastest server takes $T$ time units to satisfy a particular request and that $N$ servers will try to respond. Even if the first response can be used by servers to cut off their servicing of the request, a total of $T * N$ time units may be spent system wide in the computation of a result which actually required only $T$ time units. The actual time "wasted" is a function of the number of cache locations and the time required to service the request (i.e., $T$).

The replication locations of a particular object are control by one of several active and passive schemes. All are based on a notion of *value* or worth. Each object replica has a local and global value which roughly relates to how many times it is accessed locally and remotely (respectively). Objects with high local values should be kept locally. Ones with high global value should be replicated on other servers to spread the access load.

Still, such schemes are limited by the high cost in some systems of broadcast communication. ROMANCE [Rodrigues and Verissimo, 1993] introduces the notion of groups for managing replica consistency and communications. A group boils down to a multicast sets (i.e., limited broadcasts).

## 3.3 Migration and argument passing

Schemes for locating an object were found to be surprisingly similar. Almost all are based on some notion of indirection with dynamic chain shortening. Fowler [Fowler, 1985] suggests a mechanism whereby object spaces contain *hints* as to the location of some remote object. These hints are followed until the object is found or an exhaustive (i.e., broadcast) search is required. When the object finally is located, all spaces traversed are updated to contain the newly discovered location as the hint for the target object. This technique has proven to be quite effective and variations are used in [Jul et al., 1988, Rinard et al., 1993, Chandra et al., 1994a, Maisonneuve et al., 1992] to name but four.

We found argument passing techniques to vary widely throughout existing systems but generally fall into one of four categories: 1) call-by-value, 2) call-by-reference, 3) call-by-visit or 4) call-by-move. 1 and 2 are encountered frequently in conventional programming languages and are available in most distributed systems. 3 and 4 are somewhat unique and are supported by systems like Emerald [Jul et al., 1988]. Both involve migrating the object to the remote site. Call-by-visit moves the object only until the message processing is complete at which time it returns to its original location. Call-by-move however leaves the object in its new location. Of course, none of these mechanisms if optimal in all cases. Much of the work in distributed systems has been related to the use (how and when) of these mechanisms.

All systems contain similar schemes for marshaling and unmarshaling messages and objects which are passed between object spaces. Objects are translated into some canonical form, transported and then reconstructed using local structures. Network objects and Pickles [Birrell et al., 1993] is fairly typical of these systems.

Proxies are used heavily throughout object-oriented distributed operating systems. They vary from simple message forwarding points [Shapiro et al., 1989] to so-called *smart proxies* [Maffeis, 1993] which incorporate some amount of local caching, control or emulation of the remote object. The main role of a proxy is to make transparent the fact that the object it represents is remote relative to the location of the proxy. In this way programmers need not worry about object location and need only one language mechanism for communicating with other objects. They also allow the system designer to transparently implement replication and migration schemes.

Most systems which support migration migrate only objects. Hsieh proposes the notion of *computational migration* [Hsieh et al., 1993]. That is, rather than performing remote accesses by RPC or migrating the remote object to be local, portions of the accessing thread are moved (i.e., migrated) to the remote site. As such, these so-called remote accesses are actually performed locally with respect to the executing thread. The benefits of such a mechanism are diminished somewhat in an object-oriented environment as a suitable level of remote object interface abstraction can significantly reduce the number of actual remote accesses via combination.

Nonetheless, a similar idea can be applied to object groups which do not share a common interface. By detecting sequences of remote accesses to a group of objects which happen to be co-located we can migrate the executing thread to the remote location and thus eliminate many of the remote accesses.

## 3.4 Meta-level support

A recent trend in distributed systems has been the separation of object definition and distribution mechanism and their recognition as orthogonal concepts. Previously programmers were forced to cope with distribution issues in special case ways which affected the way they designed their application components. For example, Eden [Almes et al., 1985] had two distinct models and mechanisms for local and distributed objects. While we agree that these different models may be required, we also agree with the designers of Emerald that they should be hidden from the programmer.

The design of meta-level support for object distribution has been approached in a number of ways. One of the more general is that found in Okamura's work with AL-1/D [Okamura and Ishikawa, 1994]. They use the AL-1/D meta-level to describe and dynamically control object location, object attachment and argument passing. Through the use of some simple applications they show that such meta-level manipulations need not be inefficient.

In a similar vein, Aksit proposes the notion of Abstract Communication Types (ACTs) to model frequent interaction types [Aksit et al., 1993]. Using Sina's composition filters, incoming and outgoing messages are captured and made into first-class objects (i.e., reified). The messages are then passed to operation-specific handlers. For example, Active Send Types for sending and Active Receive Types for receiving.

The reification of these operations gives distributed systems programmers a place to stand and implement remote object references, object migration and replication, and various schemes for argument passing. Since these are all meta-level issues and are implemented in concrete meta-level, they do not impact on the application programmer or her code. This is a common theme in meta-architectures for distributed systems.

GARF [Garbinato et al., 1993] contains a limited, two-level meta-architecture based on *encapsulator* objects which intercept messages to their referent and perform transformations etc. XENOOPS [Bijnens et al., 1993] uses reference objects to implement a similar technique and give control over four aspects of the meta-level: location, migration, granularity and replication. They identify a hierarchy of reference types including; remote, local and a two references to replicated objects.

Krueger [Krueger et al., 1993] details a meta-level architecture for building user-defined virtual memory manager policies. An underlying kernel provides basic paging primitive and the meta-level defines their use policies. This is a low-level page-based approach which may not be suitable for finer-grained distributed object systems.

# 4 Applications

We were most interested in work related to the interaction between applications and distribution mechanisms. There were few papers which were explicitly about multi-space (distributed) applications. Most of the example applications came in the form of test cases for the various distribution techniques.

In general, these were not really "applications" but rather, "algorithms" (e.g., Travelling Salesman, test case generation, etc.) The algorithms are often described as irregular symbolic computations. And indeed, they did involve varying degrees of computational or communicative irregularity. Few however exhibited very much object fragmentation. Unfortunately, this is the focus of our work.

Take for example, the N-body problem [Barnes and Hut, 1986]. The problem itself has an inherent degree of interactive irregularity in that object (i.e., particle) interaction partners and physical location vary with time. Left unchecked, this irregularity would surface in the computational and communicative properties of an algorithm and adversely affect its performance. In an effort to avoid this, algorithm designers typically exploit some aspect of the implementation (code or machine) to instill and spread regularity.

In the 2D N-body problem they might use a distributed quad-tree data structure to hold the particles. The key objects in the algorithm are no longer the irregular unstructured objects (particles) but the well-structured tree nodes. The solution to the N-body problem, an irregular symbolic problem, comes in the form of a quad-tree decomposition algorithm which is somewhat more regular. Especially with respect to object (i.e., node) fragmentation.

Of course, this technique is very important and widely applicable. It does not however, result in the sort of object graph for which we are looking.

## 4.1 Reviewed applications

Below we enumerate some of the applications we found interesting and summarize their properties in our domain.

**Traveling Salesman (TSP)** [Bal and Kaashoek, 1993, Tanenbaum et al., 1993, Burdorf, 1993] The common approach to TSP uses a branch and bound implementation. Prospective paths are distributed among available processors and a global shared upper bound on the path length is maintained. This application is nice in that it is simple to implement and understand the shared behaviour. By the same token, it is quite simple and not particularly representative of the type object interaction for which we are looking.

**Gröbner basis** [Chakrabarti and Yelick, 1993, Siegl, 1993] The Gröbner basis is commonly used in the solution of problems in polynomial idea theory and represents a very important symbolic application of parallel/concurrent computing. The Buchberger [Buchberger, 1985] algorithm is perhaps the most commonly used. It reduces roughly to Gaussian elimination were the basis is analogous to a triangular set of linear equations.

Details of the algorithms are beyond the scope of this document however, we do summarize typical implementations. Loosely speaking the computation is pipelined with input polynomials entering and flowing through a set of filters which test, modify and perhaps cancel the polynomials. Results from one stage (filter) passed to the next until the end of the pipeline is encountered. Depending on the state of the polynomials, they maybe sent back to the beginning for further processing.

Generally there are only a few data structures (objects) required; a priority queue for storing potential basis elements and the basis itself. The basis can either be partitioned or replicated. Replication requires more memory overall but results in more parallelism and reduced communication overhead.

While this problem is computationally irregular in that the amount of computation done varies, it is not particularly irregular with respect to object interaction. There are two major data structures and they are pretty much global. Like the N-body problem however, it is a simple test case which should be easy to implement and test.

**Successive over relaxation (SOR)** Many problems can be solved using grid-based or stencil algorithms. Successive over relaxation (SOR) [Feeley and Levy, 1992, Krueger et al., 1993, Chase et al., 1989] is one example. The goal of SOR is to somehow interrelate (e.g., average, apply Laplace equations) neighboring cells in a data structure until a steady state is reached.

The main data structure is of course the array or collection of cells. In parallel systems the array is partitioned over the available processors and the computation in each partition proceeds more or less autonomously. The main issue is how to deal with the cell overlap. For any given cell there will generally be four others which require its value in order compute their new value. In this way, the problem really boils down to replication techniques on static (in form) data structures. This limits our interest in the application.

**WaToR [Fox et al., 1988]** WaToR, aka "the sharks and fishes" problem, is similar to SOR problem but can be made more interesting. The base problem is to model predators and prey in an ocean environment. If all the fish and all of the sharks follow the same behaviour then it is very much the SOR problem because the data elements (i.e., creatures) are interchangeable. This makes the system quite regular.

If however, each creature had unique behaviour and carried with it some amount of history etc., they would no longer be interchangeable and the problem becomes much more interesting. The creatures are transformed from data to objects. The appeal of this application is both its simplicity and its potential for complexity — We can make it as simple or as complex as we like.

**Real root isolation** [Siegl, 1993] A divide and conquer technique can be used to solve the real root problem. The divide and conquer technique leads to a tree of search processes where only the results of some paths will actually be required. To eliminate this waste, the search results are returned as something akin to a future or a suspended process. The decomposition can then ensure that only those results actually required computed. This problem displays a significant amount of regularity and as such is not relevant to our work.

**Shortest path** [Bal and Kaashoek, 1993, Burdorf, 1993] This is a fairly standard problem having a number of different algorithms. All see to display a significant amount of regularity and homogeneity of object interconnection and as such are not particularly relevant here.

**Neural net training** [Tanenbaum et al., 1994] Neural net training is a very time consuming task. Especially when the number of patterns to be recognized is large. Typical training algorithms apply a pattern to the input neurons, and then propogating back from the outputs some measure of the error. The internal weights are then adjusted and the process repeated for a different pattern. When all patterns have been input once (or some small number of times), the whole cycle is repeated. Tanenbaum proposes a parallel trainer which concurrently applies all patterns to replicas of the network and then updates the weights of the replicas. Since weight update is simply addition, the update ordering is irrelevent.

This is a useful application but rather limited in our context because of its high regularity and limited number of objects.

**Computer chess** [Tanenbaum et al., 1994] Tanenbaum has described a computer chess algorithm which essentially reduces to an alpha-beta search. As with most of the Orca test applications, it is interesting but displays considerable amount of regularity in terms of the object interaction graph.

**Test pattern generation** [Tanenbaum et al., 1994] The problem of generating a set of test patterns (inputs) which will test all circuits in a given gate array is very important in electrical engineering. Unfortunately is it NP complete. Tanenbaum proposes a parallel implementation of a pre-existing generator. The shared or distributed objects mainly consist of data structures to maintain lists of found patterns. Processors share these lists to ensure that they do not duplicate work.

**Splash** [Singh et al., 1992] A collection of parallel benchmarks from Stanford. Many have already been outlined here. They are predominately scientific and engineering algorithms of one sort or another. See reference for more details.

**OO1 benchmark** [Cattell and Skeen, 1992] The OO1 benchmark is designed to help people assess the performance of various distributed database systems for engineering applications (e.g., CAD). They provide a simple set of measures for operations which they feel are typical within their domain. In particular, lookup, traversal and insertion. The benchmark is very simple and as such it can be easily implemented in a wide range of database environments and computer languages.

Our interest in this benchmark suite is that it is a model of engineering applications. They have determined the kind of data manipulations which frequently occur. Unfortunately, though they provide some insight, we don't know how the data is typically related. This may not be very difficult to surmise however.

**HyperModel** [Anderson et al., 1990] The role of database systems in engineering applications is growning. Correspondingly, system designers require ways of measuring database performance. The OO1 (see above) and HyperModel benchmarks are attempts at addressing this need. The HyperModel benchmark suite specifies a number of operations (e.g., lookup, insertion, traversal) which should be performed on a reasonably regular hypertext structure. The focus is on assessing the performance of the database system in facilitating these operations. The benchmark as it stands involves only one data server and one accessor. As such, no elements of distribution (e.g., replication, migration) are discussed.

This would be an interesting direction in which to go because it has a concrete real-world base and is seemingly easy to describe. The benchmark code is apparently publicly available.

**Persistent store model** [Ghandeharizadeh and DeWitt, 1994] Ghanderharizadeh proposes a multi-key database partitioning scheme (MAGIC) for using more than one key to split data over a set of processors. While this is not directly relevant, the general area contains some interesting ideas. One possibility is to use an MPP architecture as a testbed for persistent object stores. Some of the processors to model the stores and some to model the accessors. This is somewhat low-level however because we have not said anything about the access patterns or what is done with the data once accessed.

**B-tree** [Hsieh et al., 1993] Hsieh uses a B-tree application to test their computation migration scheme against RPC. The B-tree itself is quite usual though it does not support deletion. In some of the tests root node replication was allowed. Though computational migration is itself interesting, this application is relatively uninteresting in the absence of interesting accessing patterns.

**Aksit** [Aksit et al., 1993, Aksit and Bergmans, 1992] briefly discusses a number of applications done with their Sina/ACT system. Many look quite interesting and applicable. Unfortunately, the material on these applications is not readily available. We are investigating which applications are directly relevant here and will attempt to obtain details.

**Hypertext systems** [Saltzer, 1993] The most attractive feature of hypertext as a distributed application is that the data is so readily distributed. Hyperlinks already form the basis for the definition and maintenance of the distributed data structure. In many ways the mail system/diary manager [Jul, 1993, Garbinato et al., 1993] applications are are constrained hypertext systems. Both suffer from the same problem: there is generally no inherent computation or efficiency issue. They do however provide good tests for object migration mechanisms but not particularly for policies. See the discussion below of our proposed hypertext-like application.

**Manager/Worker UI** [Siegl, 1993] Siegl proposes using a fairly standard worker/server architecture to handle a concurrent user interface. This application would be relevant to our domain if we had a set of interactions generated by the user which resulted in interesting interactions between the workers and servers.

# 5 The way ahead

Having reviewed the current literature in this area, we now proceed to plot the way ahead. The design distributed object systems appears to have been extensively researched but to date these techniques have been applied to applications small-scale (in terms of disjoint object spaces) with relatively regular fragmentation behaviour. In view of this, we propose to draw on the sets of available techniques and apply them to applications involving hundreds to thousands of object spaces and diverse fragmentation behaviours.

## 5.1 Mechanisms

Meta-architecture based mechanisms are the only reasonable environment in which to approach the problems of distributed computing. Several of the systems we reviewed were supported by underlying meta-architectures of varying degrees of sophistication. We will use a fully general meta-architecture which will allow the monitoring and modification of all aspects of object behaviour. Though we are primarily concerned with those aspects which pertain to distribution, the implementation of many of the schemes outlined above may involve many seemingly unrelated aspects. For example, we may wish to base replication policies on information about object performance gathered dynamical.

Our experiments will concentrate mainly on observing the relationships between application object behaviour (i.e., interaction) and the mechanisms used for message argument passing, migration and replication. For message argument passing we will consider all of the techniques available in Emerald. That is, pass-by-value, pass-by-reference, pass-by-visit and pass-by-move. These will be combined with various replication schemes chiefly ones involving eager and lazy updates and group-based replication. Migration techniques for both data and computation will also be incorporated.

We must take great care in conducting the experiments so as to isolate the effects of the different schemes and thus allow us to draw useful conclusions. As the experiments progress we expect to gain insight into the likely effects of particular changes and will be able to direct our effort in the most promising directions.

## 5.2 Applications

We would like for the usefulness and relevance of our applications to be self-evident. However, it must be noted that our goal is to model the object fragmentation behaviour of these applications, not the actual computation. So for example, if we were to model a banking system, the relevant information is that some transaction occurs over some set of data. The content or accuracy of that data is irrelevant. This frees us somewhat in that we do not have to create a real up and running banking system. Of course, we must also take great care in ensuring that the absence of real content does not somehow affect the way objects interact in the real system. This might happen if particular bank accounts are often overdrawn and so are involved in more transactions.

We have reviewed the applications used in the literature and consulted with a number of researchers in the area of object-oriented distributed and parallel computing in our search for suitable, illuminating applications. The applications with which we intend to experiment are enumerated below. Further work is needed to better define the nature of these applications but for now they provide rough guidance.

**N-body** The N-body problem [Barnes and Hut, 1986], while relatively simple and perhaps not entirely illuminating should prove to be a good baseline for comparison of our work to that of others. That is, we can use it in an attempt to duplicate other's results and thereby gain confidence in our experimental procedures and testbed. Additionally, it is quite simple and so will require little effort.

*Tj* **A** There is a broad range of problems which might best be described as integrated intelligent manufacturing or business enterprises. We use an airline as an example but do not focus on the reservation system but rather the operational side. Below we give a brief description of the real-world situation followed by summary of the relevant points for our modelling.

An airline, *Tj* Airlines (*Tj*A), which has several hundred aircraft flying routes throughout the world. As per normal airline operations, *Tj*A has several regional hubs and service centers. Planes are crewed by qualified trained and available people on routes for which they have been approved and serviced by technicians who are qualified for particular tasks. Local arrangements must be made at every stop for fuel, gate clearance, cleaning, boarding, flight plans, etc.

To make the system more realistic we must also consider that flights are delayed, planes break, crews and staff become unavailable due to illness or delays in previous flights and airports close due to weather. These changes in the "normal" or scheduled operation ripple through the system and are a source of non-determinism.

In general, the computational model for solving or implementing this application is a constraint solver. That is, if all you are concerned with is the scheduling. However, the real system is more complicated because events and data must be tracked and used in computations beyond scheduling; inventory, payroll, planning, billing and maintenance to name but a few.

While planes and crew (i.e., resources) travel from region to region, they are always member or part of their home region. Within the system, resources can be in several systemic regions at once. A plane is going from A to B and is currently at C but is scheduled for maintenance at D etc. Resources can be acquired, consumed, lent or transferred. As they move through and exist in the system the accumulate connections or relationships to other resources as do other resources with them. These relationships can be transient or permanent as required.

Regional centers have various computational requirements and resources. Regional computing may involve objects residing in other regions, foreign objects which are temporarily local (via replication or visitation) and of course, local objects. The computations themselves range from simple data accesses to complex planning/forecasting to lengthy transactional processing.

At the ultimate level, we consider each aircraft's avionics and computer systems as part of the airline's distributed computer. It can interact with ground-based systems either via direct cable links while on the ground or via radio telemetry while in flight.

With enterprises such as this we could go on and on, diving deeper into detail but there really is no need. Even with this relatively superficial model we have a rich set of interactions and interdependencies which should make for quite dynamic object fragmentation behaviours.

**Hypertext** In considering the fundamental requirements of our applications, diverse data interconnection was one of the key points. This phrase is a very apt description of typical hypertext systems. Unfortunately, hypertext systems are typically fetch-oriented. That is, they are essentially distributed databases with no particular computational aspect. Other than caching, there is no need for objects to be replicated or migrate.

There are several approaches to this problem. One is to have some static set of predicted operations which are generally representative of some domain's use of hypertext. For example, the HyperModel benchmark [Anderson et al., 1990] (see description above). This general idea is acceptable but as we are interested in object interaction rather than simple fetching, we must increase the number and sophistication of the benchmark's closure traversal operations to utilize insert, delete and edit operations thus modifying the object interconnection graph.

Similarly we could introduce agents into the system. An agent is simply computation or access and change. We can envisage a number of different kinds of agents with tasks like searching, destroying, organizing, synthesizing, etc.

While this model is attractive and relevant, it is not clear if we can find a sound basis on which to model our agent's behaviour. We believe that there are few such existing systems. One option may be to model the agents and data after some real-world situation. We would however have to ensure that something similar to the real-world system is natural and desirable in the computer domain.

One possibility is to gather data from the ANU Pastime system. Pastime is a project to build a hypertext/multimedia system for the Australian Parliament. It is quite a large system incorporating the Parliamentary Library, video and audio libraries as well as all of the Parliament's legislative material. Access terminals will be available to the public and members of Parliament and other government officers.

If we could monitor and perhaps characterize the usage patterns - how and what people look at/for - we could then model the people as active agents in the hypertext system. Unfortunately it is unlikely that any one person will do much that is interesting simply because of the time and patience required. This is the vary reason for having active agents in the first place. The ideas in the HyperModel benchmark may be of significance here. At any rate, more investigation will be needed before we can determine the feasibility of this application.

**Cooperative work environment** Somewhat related to the idea of active agents in a hypertext system, we can consider a system for Computer Support of Cooperative Work (CSCW). More work will have to be done to investigate the nature of the object interactions.

## 6 Conclusions

This should be real fun!

## References


[Aksit and Bergmans, 1992] Aksit, M. and Bergmans, L. (1992). Obstacles in object-oriented software development. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 27, pages 341–358.

[Aksit et al., 1993] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A. (1993). Abstracting object interactions using composition filters. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 152–184. Springer Verlag.

[Almes et al., 1985] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. (1985). The eden system: A technical review. *IEEE Transactions on Software Engineering*, 11(1):43–58.

[Anderson et al., 1990] Anderson, T. L., Berre, A. J., Mallison, M., Porter, H. H., and Schneider, B. (1990). The HyperModel benchmark. In *Proceedings of the Conference on Extending Database Technology*, LNCS 416, pages 317–331. Springer Verlag.

[Bal and Kaashoek, 1993] Bal, H. E. and Kaashoek, M. F. (1993). Object distribution in orca using compile-time and run-time techniques. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 28, pages 162–177.

[Bal et al., 1992] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. (1992). Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205.

[Barnes and Hut, 1986] Barnes, J. and Hut, P. (1986). A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449.

[Bennett et al., 1990] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. (1990). Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, volume 25, pages 168–176.

[Bershad et al., 1988] Bershad, B. N., Lazowska, E. D., Levy, H. M., and Wagner, D. (1988). Preseto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8).

[Bershad et al., 1993] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A. (1993). The Midway distributed shared memory system. In *Proceedings of CompCon '93 (IEEE)*, pages 528–537.

[Bijnens et al., 1993] Bijnens, S., Joosen, W., and Verbaeten, P. (1993). A reflective invocation scheme to realise advanced object management. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 139–151. Springer Verlag.

[Birrell et al., 1993] Birrell, A., Nelson, G., Owicki, S., and Wobber, E. (1993). Network objects. In *Proceedings of the Symposium on Operating System Principles*, volume 27, pages 217–230.

[Buchberger, 1985] Buchberger, B. (1985). Gröbner basis: An algorithmic method in polynomial ideal theory. In Bose, N. K., editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. D. Reidel Publishing Company.

[Burdorf, 1993] Burdorf, C. (1993). A performance evaluation on caching mechanisms for a persistent object system. *ACM SIGPLAN OOPS Messenger*, 4(3):41–49.

[Cahill et al., 1993a] Cahill, V., Baker, S., Horn, C., and Starovic, G. (1993a). The Amadeus GRT – Generic runtime support for distributed persistent programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 28, pages 141–161.

[Cahill et al., 1993b] Cahill, V. et al., editors (1993b). *The COMMANDOS Distributed Application Platform*. Springer-Verlag.

[Cattell and Skeen, 1992] Cattell, R. G. G. and Skeen, J. (1992). Object operations benchmark. *ACM Transactions on DataBase Systems*, 17(1):1–31.

[Chakrabarti and Yelick, 1993] Chakrabarti, S. and Yelick, K. (1993). Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, volume 28, pages 169–178.

[Chandra et al., 1994a] Chandra, R., Gharachorloo, K., Soundararajan, V., and Gupta, A. (1994a). Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 274–288.

[Chandra et al., 1994b] Chandra, R., Gupta, A., and Hennessy, J. L. (1994b). COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26.

[Chase et al., 1989] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. (1989). The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Symposium on Operating System Principles*, pages 147–158.

[Dickman and Makpangou, 1992] Dickman, P. and Makpangou, M. (1992). A refinement of the Fragmented Object model. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 230–234.

[Feeley and Levy, 1992] Feeley, M. J. and Levy, H. M. (1992). Distributed shared memory with versioned objects. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 247–262. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[Fowler, 1985] Fowler, R. J. (1985). *Decentralized object finding using forwarding addresses*. PhD thesis, University of Washington. Also available as Dept. of Computer Science Tech Report 85-12-1.

[Fox et al., 1988] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. (1988). *Solving problems on concurrent processors*, volume 1, chapter 17, pages 307–325. Prentice Hall.

[Garbinato et al., 1993] Garbinato, B., Guerraoui, R., and Mazouni, K. R. (1993). Distributed programming in GARF. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 225–239. Springer Verlag.

[Ghandeharizadeh and DeWitt, 1994] Ghandeharizadeh, S. and DeWitt, D. J. (1994). MAGIC: A multi-attribute declustering mechanism for multiprocessor databse machines. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):509–524.

[Hosking and Moss, 1993] Hosking, A. L. and Moss, J. E. B. (1993). Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Symposium on Operating System Principles*, volume 27, pages 106–119.

[Hsieh et al., 1993] Hsieh, W. C., Wang, P., and Wiehl, W. E. (1993). Compuation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, volume 28, pages 239–248.

[Jul, 1993] Jul, E. (1993). Separation of distribution and objects. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 47–54. Springer Verlag.

[Jul et al., 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

[Krueger et al., 1993] Krueger, K., Loftesness, D., Vahdat, A., and Anderson, T. (1993). Tools for the development of application-specific virtual memory management. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 28, pages 48–64.

[Liskov, 1988] Liskov, B. (1988). Distributed programming in argus. *Communications of the ACM*, 31(3):300–312.

[Maffeis, 1993] Maffeis, S. (1993). A flexible system design to support object-groups and object-oriented distributed programming. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 213–224. Springer Verlag.

[Maisonneuve et al., 1992] Maisonneuve, J., Shapiro, M., and Collet, P. (1992). Implementing references as chains of links. In *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 236–243.

[Nitzberg and Lo, 1991] Nitzberg, B. and Lo, V. (1991). Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60.

[Okamura and Ishikawa, 1994] Okamura, H. and Ishikawa, Y. (1994). Object location control using meta-level programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, pages 299–319. Springer Verlag.

[Pu et al., 1992] Pu, C., Florissi, D., Soares, P., Wu, K.-L., and Yu, P. S. (1992). Performance comparison of active-sender and active-receiver policies for distributed caching. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 218–227.

[Rinard et al., 1993] Rinard, M. C., Scales, D. J., and Lam, M. S. (1993). Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38.

[Rodrigues and Verissimo, 1993] Rodrigues, L. and Verissimo, P. (1993). Replicated object management using group technology. In *Proceedings of the Workshop on Future Trends of Distributed Computing Systems*, pages 54–61.

[Saltzer, 1993] Saltzer, J. H. (1993). Needed: A systematic structuring paradigm for distributed data. *ACM SIGOPS OS Review*, 27(2):77–81.

[Shapiro et al., 1989] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M., and Valot, C. (1989). SOS: An object-oriented operating system – Assesment and perspectives. *Computer Systems*, 2(4):287–337.

[Siegl, 1993] Siegl, K. (1993). Parallelizing algorithms for symbolic computation using ||Maple||. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, volume 28, pages 179–186.

[Singh et al., 1992] Singh, J. P., Weber, W. D., and Gupta, A. (1992). Splash: Stanford parallel applications for shared memory. *Computer Architecture*, 20(1):5–44.

[Tanenbaum et al., 1994] Tanenbaum, A. S., Bal, H. E., Hassen, S. B., and Kaashoek, M. F. (1994). Object-based approach to programming distributed systems. *Concurrency: Practice and Experience*, 6(4):235–249.

[Tanenbaum et al., 1993] Tanenbaum, A. S., Bal, H. E., and Kaashock, M. F. (1993). Programming a distributed system using shared objects. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 5–12.

[Tanenbaum et al., 1990] Tanenbaum, A. S., van Renesse, R., van Straveren, H., Sharp, G. J., Sape J. Mullender, J. J., and van Rossum, G. (1990). The amoeba system. *Communications of the ACM*, 33(12):46–63.