Run-Time Support and Storage Management for Memory-Mapped Persistent Objects

Bruce R. Millard — Arizona State Univ., Tempe, Arizona

Partha Dasgupta — Arizona State Univ., Tempe, Arizona

Sanjay Rao — Tandem Computers, Cupertino, California

Ravindra Kuramkote — Univ. of Utah, Salt Lake City, Utah

Abstract

Conventional object-oriented programming systems allow application programmers to structure each application as a set of objects. They do not allow long-term storage of the objects, nor do they allow sharing and concurrency within the object spaces. Persistent object systems and object-oriented databases have been developed to address some of these shortcomings. Persistent object systems provide the ability to store, share and dynamically reuse the object instances and object structures that the application creates or uses.

A persistent object system depends upon a store (on secondary memory) to provide the persistence. In addition the store is responsible for the provision of services such as object identity or naming, type security, retrieval, sharing support, locking and concurrency control, consistency control and other features necessary for the management of long-lived objects.

Persistent stores have used many techniques to provide the above. The two most common are the object-oriented database and the specialized object-container based stores. We are investigating the use of memory mapping combined with a storage manager to provide a distributed object store. This paper presents the design and implementation of SPOMS which is a memory-mapped store built on top of the Mach operating system.

Introduction

Conventional object-oriented programming systems [Co86] do not provide adequate support for storing and sharing the objects and data types used in application programs. Persistent Object Systems are an extension that adds the features of persistence and sharability to language defined objects [An91]. In particular, the concept allows graphs of objects to be stored and retrieved transparently, and allows these object structures to be exchanged and used concurrently between separate applications. These features are generally provided by the use of files coupled with "pointer-swizzling" techniques or by the use of Object-Oriented Databases. Operating systems and architectural support for persistent objects have also been studied.

One method of storing and using objects is to use memory mapping. A persistent store comprises of a set of objects that are mapped into the address space of an application upon demand. Memory mapping makes use of operating system support and makes the storage and manipulation of persistent object somewhat simpler. Also, this technique can provide sharing at a much finer grain than what is provided by database systems.

This paper presents the design and implementation of a persistent store called SPOMS. SPOMS is a runtime system that provides a store for persistent objects and is language independent. The objects are created via calls to SPOMS and when used, SPOMS directly maps them into the spaces of all requesting processes. The objects are stored in native format and are concurrently sharable. The store can handle distributed applications.

The system uses the concept of a compiled class to manage persistent objects. The compiled class is a template that is used to create and store objects in a language independent manner and so that object reuse can occur without recompilation or re-linking of an application that uses it

A prototype of SPOMS has been built on top of the Mach operating system. The paper presents the motivations, the design, implementation details and also contains a discussion of related and future work.

Related Work

Simply put, object-oriented programming allows programmers to deal with program structuring units called objects. Each object is an instance of a class. A class defines the code and data contained in the object or instance. The advantages of the programming paradigms that use the object-oriented concept are relatively well known and widely accepted.

With the maturing of object-oriented programming systems, a few important shortcomings have been noted. One of the primary concerns is the inability (in most language based systems) to elegantly store, re-use and share instances of objects that are created during the execution of an application. Persistent object systems have addressed this problem in a variety of ways.

Early work involved using files as **object holders** and providing the programmer with routines to load and store objects from the files. *Eiffel* [Me92] allows a tree of objects to be copied into a file and then the file containing the saved objects to be re-loaded into memory. In the *Coral3* system [MeLa87] Smalltalk objects are stored in the file system. Locking is provided at the object holders to allow proper (controlled) sharing of persistent objects across applications. The majority of the current work involves a technique called **pointer swizzling** (or object faulting) and the use of persistent object stores or object-oriented databases.

The pointer swizzling technique [Ri90, Co*84, HoMo90, Wi90] is used by languages where all object instances are accessed through pointers, one pointer per instance. For each persistent object, the initial pointer points to a stub routine instead of the actual object. When this pointer is accessed, the stub routine gets executed. The stub routine generates a fault, causing the object to be loaded from a store (or file) and then it changes the pointer (swizzles it) to point to the actual object. Subsequent object accesses get the real object and not the stub. The pointer swizzling technique is, generally, combined with an object store or a database that actually provides the storage management. *Mneme* [Mo90] is a good example of a persistent object store and is similar to a "low-level" database. Mneme stores objects in its own format. Each object has a unique name (or id) and is stored as a vector of data and pointers. The pointers are pointers to other objects in the store, and hence are object-ids. Mneme supports locking, consistency control and distribution.

The object-oriented database is an elaborate, general purpose object store . It stores all data as objects that can be accessed and used by regular object-oriented languages (typically slightly modified C++). The application program retrieves objects from the database, processes them and stored them back into the database. The database handles object identity [KhCo86], storage management, concurrency control and transaction processing. Object-oriented databases also fits the requirements of a persistent object system. Some of the research object-oriented databases developed in recent years include *Orion* [Kim*88, Kim90], *GemStone* [Ma*86, PeSt87], *EXODUS* [Ca*89], O^2 [Ve*90], *ObServer/Encore* [Sk*86, El*89, HoZd87], *Cosmos* [Sr*91], *Iris* [Fi*89], *Zeitgeist* [Ford*88]. Object-oriented databases have also been commercialized and many off-the-shelf versions are available. These include *GemStone* [Br*89, Sr*90], *ONTOS* [Ah91], *VERSANT* [Ve92], *Orion/ITASCA*, *ObjectStore* [St88, At90].

Some operating systems implement their own versions of objects at the kernel level. These include *Argus* [Li*87], *Cronus* [Sc*86] *Eden* [Al*85] and *Clouds* [Da*91]. Systems such as the *Commandos* operating system [MaGu89] and *Guide* [Ba*91] provide extensive language-level object support via operating system routines.

Memory mapping of objects into applications spaces have been used by *Cricket*, *MONADS* and *Casper*. Cricket [ShZw90] maps the entire object store to the address space of all applications using the store. *MONADS* [Ro90 uses a large address space to structure the object store and selectively maps the needed objects into the application space. *Casper* [Va92] is a Napier88 system implemented on Mach that uses memory mapping and shadow paging to provide a distributed resilient persistent store. Mach facilities are used to map the store to the address space of the processes using the store and then external pagers are used to detect accesses to the mapped pages. The access detection can be used to provide coherence in distributed-memory consistency control via copy-on-write and commits.

Motivations and Goals

Our work was motivated by the need for a simple external store for persistent objects that is:

- language independent,
- memory mapped and
- distributed

Most persistent stores use pointer swizzling as the access mechanism and thus are tied to the language primitives and need modifications to both the language and the compiler. Having language independence would allow programmers of pre-packaged objects to write them in any language of choice (including non object-oriented languages like C). The objects can then be accessed by applications that are written in other languages via a set of runtime library routines.

Memory mapping provides a whole set of advantages. First, all objects are stored in their native format. Their is no need to convert from one storage format to another and vice versa. Second, the sharing of the objects between applications happen in real-time. That is, changes made by one application are instantly visible in another application. This can be termed concurrent sharing or fine-grained sharing (as opposed to sequential sharing or coarse-grained sharing possible in load-store or database systems). Finally, distributed shared memory systems can be used to provide distributed implementations of the store.

The Design of SPOMS

SPOMS (Shared Persistent Object Management System) is an object-oriented language support system that provides user programs with a simple, language independent, transparent interface to persistent objects. These objects are concurrently sharable and mapped to the application's virtual address space as needed from distributed persistent object stores. Persistence, sharing and demand loading are provided without changing the language's compiler.

SPOMS is a runtime storage manager as well as a set of language pre- and post- processors that create the objects to be contained in the store. Central to the design of SPOMS is the concept of a **compiled-class.**

The compiled-class is a storage unit, an intermediate-class, a template as well as a language independent entry into SPOMS. All objects stored by SPOMS must be created and manipulated though SPOMS. Before any object is created a compiled-class needs to be installed.

A programmer writes a definition and implementation of one or more objects in a file. This file is pre-processed, compiled and post-processed to yield a language independent module called the compiled-class. The compiled-class is then registered with SPOMS. Once registered the compiled-class is available for access from any application.

An application program can ask SPOMS to create an instance of any compiled class contained in SPOMS. When created, SPOMS places the an instance (or copy) of the compiled class in the address space of the application. The application can then make calls on the methods contained in the instance. This instance becomes a persistent instance and can be concurrently accessed by other applications as well as used later by the application that created it.

Memory Management and Mach

Since SPOMS manages memory-mapped persistent objects, it needs significant memory management support from the underlying operating system. The necessary features are:

- The ability to map memory segments into an application at a specified point in its address space.
- The ability to protect a part of the memory segment from access by the application which has mapped it (for locking control).
- The ability to be informed if an application page-faults on a memory segment controlled by SPOMS.

Mach [Ac*86, Ra*87] provides all the underlying mechanisms needed to provide shared, persistent objects that are accessible on demand at run-time and in addition has some other interesting programming features, e.g., kernel threads and an expressive IPC. The Mach approach to virtual memory management [Yo*87, Yo*89] permits local, single-copy sharing of code and data, object faulting and transparent on-demand object access, and is implementation extensible to a distributed environment..

Mach virtual memory management specifics of interest to this research are embodied in the *memory-object* and the *External Memory Management Interface* (EMMI). A memory-object is an object on secondary storage that can be mapped to the virtual address space of a task. It can be implemented by any object that has a port and that can read and write data. The memory-object mapping routines allow the protection associated with a memory-object to be changed in ways that permit the locking of shared objects.

The Mach EMMI allows user-level server programs called *external pagers* or memory managers to manage objects that can be mapped into the virtual memory of a task. These servers can manage permanent storage, consistency, and sharing of data objects between tasks. The basic abstraction used by the EMMI is the memory-object described above. EMMI allows the operating system to access data represented by the memory object and leaves the control of data usage to the memory manager.

The memory manager allows a user to create memory-objects that are nothing but chunks of memory identified by ports. A Mach thread maps such objects to its task's address space to access the memory object. Once the object is mapped, page faults on this object are sent by the kernel to the port identifying this object which are, thus, received by the memory manager. If threads in two tasks map the same memory object, then the kernel sends page fault requests for each page only once when either of the threads touch the page. Consequently, it becomes the responsibility of the memory manager to maintain consistency of these pages using appropriate locking.

SPOMS is implemented as a set of cooperating memory-objets and external pagers. To provide distribution the external pagers need to be programmed to allow coherence between sites using a scheme similar to the Mach NetMemoryServer [Fo*88]. The NetMemoryServer is an implementation of Distributed Shared Memory on Mach that provides multiple-reader and single-writer coherency between memory-objects on a network of computers.

SPOMS Architecture

SPOMS has three major subsystems: the language specific Compilation Subsystem, the Object Management Subsystem, and the Runtime Subsystem. Users interact with the Compilation Subsystem converting class definitions (source code) into compiled-classes (compiled and partially linked object code). Users also employ the Compilation Subsystem to create object instantiations. Lastly, the Compilation Subsystem compiles and links applications that reference SPOMS objects and compiled-classes. The Compilation Subsystem relies on information maintained by the Object Management Subsystem to complete its work. The Object Management Subsystem stores compiled-classes and objects, keeps track of related information (such as entry points, size, authorizations), and manages the capabilities that users utilize when referencing objects and compiled-classes.

The SPOMS Runtime Subsystem responds to application object faults on persistent objects (either data reference faults or initial method references via the stub), compiled-class references for dynamically loading transient compiled-classes, and compiled-class references for dynamically creating sharable, persistent object instances. The Runtime Subsystem passes the capabilities passed by the application on object or compiled-class references to the Object Management Subsystem to map

SPOMS objects into the application's virtual memory. In the case of compiled-classes, the runtime system also needs to link/load the code associated with its methods, if any.

Design Issues

Now that we have presented the motivation and an overview of SPOMS there are a few design issues that effect the design and implementation that should be discussed. These issues are:

- What is concurrent sharing and what methods are available to achieve it?
- What methods are available to implement persistence?
- How can dynamic loading (runtime access) of objects be done?
- Are there solutions to the above that admit "simple" solutions to providing a distributed SPOMS?

This section will address these questions in the above order, saving most distributed operation issues to last.

Concurrent Sharing

Concurrent sharing in SPOMS means that the methods associated with an object can be executed concurrently. It also means that data associated with a SPOMS object should be concurrently available to those processes using the methods or, in the case of pure-data objects, directly by the referencing object, i.e., processes or SPOMS objects. This last property requires some form of concurrency control policy. The first property, on-the-other-hand, necessitates re-entrant code. Since one of the reasons for providing concurrent sharing of objects is to provide single (local) copies of objects, the underlying system should support simultaneous access to a single copy of object code and data.

Concurrency control policies can be viewed either from the user process perspective or as seen by the system. From the process view the programmer asks, "What limitations are placed on free access to the data?" or "How transparent is the access?" On-the-other-hand, the system implementer must know whether to provide single-writer/multiple-reader coherency, multiple reader/writer, or none at all. For this part of SPOMS, we chose to ignore coherency at the local system level because the single copy implementation should make this feasible and because this approach will provide the greatest flexibility to the application and object programmers. Initially, it was felt that concurrency control could be supplied entirely by the programmer. However, there are difficulties to supplying a set of generic, user-level, concurrency control primitives, which are even worse in a distributed environment, and so eventually the system, either SPOMS or its host environments, will need to supply such primitives.

The mechanisms necessary to support shared code or data ultimately must be provided by the host operating system. Several sharing mechanisms exist in currently available operating systems. Available for code sharing are: dynamically linked libraries (several operating systems), memory-mapped files (some Unix operating systems), shared memory (some Unix operating systems), and the memory-object and External Memory Management Interface of Mach. Data sharing mechanisms include all of these except dynamically linked libraries. The memory-mapped file and shared memory implementations in current Unixes make it difficult (and usually impossible) to do object faulting. Without object faulting all shared objects referenced by a process need to be mapped into the process at process instantiation or extensive library support for object loading is required. In addition, these mechanisms do not scale to a distributed environment at all simply. The essentials of the Mach approach are discussed above.

Persistence

A SPOMS goal is to provide transparent persistence. Transparent persistence means that the base programming language of an object or application should not require modification to use a persistent object. Our means of implementing this are concentrated in the Compilation Subsystem. The Compilation subsystem does pre-processing of source code and post-processing of the base-language compiler output to implement language level transparent persistence.

Operating system support for persistence is also necessary. One approach to providing persistence is to use files to maintain the long-term state of persistent objects. This approach involves extensive library code for reading and updating such objects, as discussed above. The file approach also requires some additional mechanism to permit concurrent sharing of the objects. Another approach to persistence is the database approach which uses well-understood Database Management techniques. While these techniques are well understood and adequate for the purposes for which they were intended, they apply, for the most part, only to data, have reduced flexibility where sharing is concerned, and suffer from the same need for extensive library code as does the file approach.

There are two alternatives to the file and database approach: memory-mapped files and the memory-object with external pager approach of Mach. While the memory-mapped file approach appears at first glance to be the same as the file approach it embeds the sharing technique needed for SPOMS. However, the memory-mapped file approach does not permit object faulting for pure-data objects, requires a lot of library code, and is not extensible to a distributed implementation. The Mach approach can accomplish all these goals as discussed above.

Dynamic Loading

Making shared, persistent objects available on demand requires two steps: first, recognizing the demand and, second, actually making the object available. When an object's method is invoked the invocation mechanism can include code to insure the object is mapped before attempting to execute the method's code. If data only objects are considered, the recognition step must include some form of object/page faulting. The SPOMS approach to demand recognition is to provide a **stub routine** that is invoked when an object that has methods is first invoked. This stub is used to guarantee that the object is mapped to the applications address space. In SPOMS data only objects are pre-mapped to a process's virtual address space so that a reference to it will produce a "catchable" memory protection fault.

After recognizing the demand for a dynamically loaded object, the object still must be loaded. There are, however, a few complications: where to load the object, how does the application know where, what are the object's internal addressing needs, and are address modifications within the object necessary (pointer swizzling)? These questions can be answered in a more straight-forward way if they are asked as, "Is the object loaded at the same address in every application every time?" If they are loaded at a predetermined address then where to load the object is known, the application knows where the object is loaded, and the object's internal addresses are known a priori with no pointer swizzling necessary. The only problems with this approach are: how to deal with overlapping objects and is there enough virtual address space to accommodate unlimited (large) numbers of objects. To accommodate overlapping objects, object remapping would need to be done. A better approach would be to assure that all objects are loaded at their own non-overlapping virtual address. Given this solution, it is necessary to have a large virtual address space (i.e., probably 64 address bits or more).

The alternative to loading objects at a fixed address is to use position independent code (PIC) and load objects wherever space is available. The PIC approach requires PIC generating compilers. Depending on the compiler, objects may still have to be loaded at the same virtual address for all concurrently referencing applications. If this latter property is the case then object remapping (caused by overlapping objects) is still necessary. While we believe that the PIC approach is more general and, hence, preferable, the current trend to 64 bit computers [HeJo91, HePa90] makes this unnecessary. In addition, the use of fixed address objects has less run-time overhead than PIC objects.

Another advantage of using fixed addresses for loading objects is that pointers to dynamically loaded objects always point to the correct place. Many object-oriented language programs rely on pointers to objects, methods or data within other objects. The use of pointers within objects that persist, i.e., persistent pointers, adds further complexity.

Distributed SPOMS

At this time we are only considering a homogeneous distributed environment for SPOMS. Both the SPOMS Run-time and Object Management Subsystems need consideration in this regard. We assume that SPOMS objects should be accessible during execution from multiple sites. This access should be transparent and timely from the application's perspective. Distributed Shared Memory (DSM) [LiHu89, NiLo91] can accomplish this most effectively. By using an implementation of DSM that provides multiple-reader and single-writer coherency, both object code and data can be shared over the network. Other methods, e.g., RPC and explicit message-passing library code, could be used to provide runt-time object sharing. However, these alternate approaches conflict with the goal of local single copies.

The SPOMS Object Management Subsystem returns capabilities to users when a SPOMS object or compiled class is created. To distribute the Object Management subsystem functionality it is necessary to locate the object's storage site given its capability. The alternatives here are to given each SPOMS object or compiled class a network-wide unique identifier and distribute identifier-location information or embed the location information in its capability. We chose the latter approach. Not only does this approach allow an object to be located at run-time but permits SPOMS to support replicated objects transparently.

SPOMS Implementation

In this section we present a quick overview of a prototype SPOMS that has been implemented at Arizona State University. Since this implementation was a prototype, we intentionally left out some of the properties of the full SPOMS specification so that we could concentrate on essential fundamentals. Three areas left to future work deserve some mention - see the section on Future Work below for our plans for these and other areas. First, fixed, predetermined addresses for dynamically loaded objects are provided because: we had no PIC generating compiler, pre-linking is easy, and we aren't convinced position independent loading is essential. Second, the dynamic compiled-class linking and loading implementation has been left until the details of dynamic object loading are worked out. Third, the distributed Run-time and Object Management Subsystems have been left till later because they will be built from the local only versions. As a last simplification, we limited our language support to the C and C++ languages.

As depicted in Figure 1, there are five components to the prototype SPOMS:

- The user and programmatic interface.
- The Compilation Subsystem.
- The Shared Object MAnager (SOMA).
- The Persistent Object Store (POS).
- The Run-time Subsystem (EOS , et al).

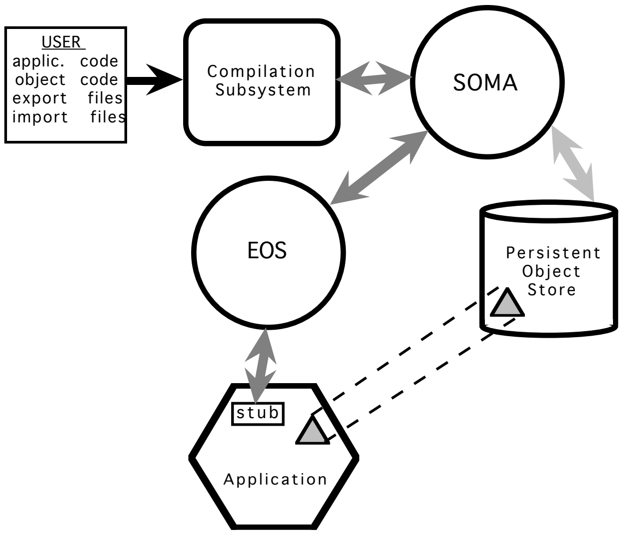

Figure 1. Prototype SPOMS Architecture.

The Programmatic Interface to SPOMS is part of the specifications of the program files passed to the Compilation Subsystem. Source code and Import files for applications and objects are needed to create compiled-classes and applications. Export files contain definitions of the methods exported by the object.

The Compilation Subsystem is responsible for generating the compiled-class and registering it with SPOMS. It compares object invocation semantics in the object source code with that in the export file and massages formal method specifications to include parameters needed by the run-time code. After compilation the number and names of methods, size of the object code, the object code is post-processed to generate the compiled-class. The complied class is then sent to SOMA using Mach IPC.

The Compilation Subsystem is also responsible for massaging the application source code to provide for the transparent, dynamic loading and, subsequent, access of SPOMS objects. This is accomplished using the Import files. Import files contain the object and method names that the application (or SPOMS object) wants to use. This information is sent to SOMA by the Compilation Subsystem prior to pre-processing to retrieve low level object and entry point identifiers. The information retrieved from SOMA is used to create a branch table. (Invocations of SPOMS objects are always made through the branch table.) The Compilation Subsystem massages the source to use the branch table. In addition to the branch table a stub routine is linked into the application for use during run-time. The Compilation Subsystem uses yacc to parse import and export files and lex to massage the source code files before compilation.

SOMA is responsible for storing compiled-classes and keeping track of the creation of compiled-classes by the compilation subsystem. SOMA also handles the naming of compiled-classes and instances. In addition to the Compilation Subsystem IPC interface, SOMA has an IPC interface to EOS for use during run-time (discussed below) and a direct-user interface for services such as: object directory lookups, object group creation and management, object access control administration, and deletion of objects. The direct-user services are built in to SOMA while the user application part of the direct-user interface is currently under development. SOMA, also, acts as a database manager for the Persistent Object Store (POS).

POS resides in the Mach file system and has three sub-components: compiled-classes, SPOMS objects and the database that describes them. Currently, compiled-classes and objects are stored independently as collections of files and the database as three others.

EOS, the External Object Server, is the essential component of the SPOMS Run-time Subsystem. EOS is an a Mach external pager. It provides the initial memory-mapping of a SPOMS object (a Mach memory-object) into the application's virtual address space. Since this initial call only provides the mapping, thereafter EOS provides any subsequent page-fault handling of references to the object.

The SPOMS Run-time Subsystem encompasses the application, the External Object Server (EOS), SOMA and POS. As mentioned the application contains library code, i.e., the branch table and the stub routine, linked in by the Compilation Subsystem. The stub routine interacts with EOS to provide demand memory-mapping of an object. During compilation/linking, the branch table is initialized to invoke the stub routine rather than an object's methods. The stub routine passes the object's id to EOS. EOS then requests object information from SOMA (if not already in use by another process), maps the object to the user's virtual address space, and returns the necessary information to the stub routine. The stub routine fixes-up the branch table so that subsequent method invocations for the object go to the object (rather than the stub) and then branches to the method code with the original request.

To accommodate a large number of applications and objects, EOS uses two groups of threads: one group handles paging activity and the other group handles user requests. In addition to the stub requests the second group handles user requests to explicitly release an object. Explicit object release is provided so that a process can reuse the space set aside for the object. Under "normal" circumstances this space remains allocated until the process terminates.

SOMA is a complicated program. It not only maintains the POS but also interacts with the user (as mentioned above), interacts with EOS at run-time, interacts with the Compilation Subsystem, manages the object and compiled-class capabilities used to access them, manages the virtual address space usage for applications and objects, and controls the coherence of the POS database. Because of the number of tasks SOMA has, SOMA uses three sets of threads to do its job. Each set of threads has a manager and some worker threads. The manager thread monitors a port associated with one of its external interfaces, i.e., user interface (UI), EOS interface (EOSI), Compilation Subsystem interface (CSI). When a message arrives at the port, the manager selects one of the worker threads to handle the details of the request. The EOSI threads are given priority over the other threads because their work involves requests form executing applications. To reduce execution time overhead the POS database has a copy both on secondary store and in memory. Since the threads access the POS database, possibly, concurrently SOMA has a set of concurrency control routines that work in conjunction with a commit control package to maintain coherence of the database. SPOMS capabilities have encoded information that is used to access the information in the database. These capabilities contain the user id of the creator or owner of the object and a low level SOMA resource identifier.

SOMA has one last function that requires special mention. In this version of SPOMS all objects are linked at a predetermined address. This address may be explicitly specified by the user or may be selected by SOMA. If the selection is left to SOMA, the user still has the option of selecting an Address Group from which SOMA selects the address. Address selection within a group occurs from the next available address for that group. The address space of an object belonging to a group is guaranteed to not overlap that of any other object in that group. This functionality was provided to allow some degree of flexibility in object placement and to attempt to reduce addressing conflicts while we only have 32 bits of address space available.

Future Work

As mentioned, three areas were left out of the prototype. As soon as we can obtain C and C++ PIC generating compilers, we will begin experimenting with relocatable objects. Relocatable objects will allow use to pursue methods of dealing with persistent pointers. Second, the dynamic compiled-class linking and loading and the effects of compiled-class inheritance are currently under study for inclusion in SPOMS. Third, distributed Run-time and Object Management Subsystems are also currently under consideration. While distributed shared memory for the run-time system seems to have a well-understood implementation, the consequences on SOMA, EOS and the need for user level synchronization are still being studied.

We also plan to extend the number of applications using SPOMS and add to the number of languages supported. Adding support for more languages will force us to address issues in heterogeneous language semantics. At this time we have no plans to include heterogeneous computers.

Conclusions

This paper has presented a runtime support system that provides user programs with a simple, language independent, transparent interface to persistent objects. The objects are concurrently sharable and mapped to the application's virtual address space as needed from distributed persistent object stores. In this system persistence, sharing and demand loading are provided without changing the language's compiler.

The system's primary purpose is to provide an environment that has extensive facilities for object reuse and distributed object sharing. SPOMS extends the concept of code reuse to data and to the concurrent sharing of compiled code and persistent data in a distributed environment. Object reuse, thus, provided without requiring recompilation or re-linking of an application that uses it. A new stage of object (called a compiled-class) was also described.

References

[Ac*86] Acceta, M., R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation For UNIX Development," *Proceedings of Summer Usenix*, July 1986.

[Al*85] Almes, G., A. Black, E. Lazowska and J. Noe, "The Eden System: A technical Review," I*EEE Transactions on Software Engineering*, SE-11, January 1985.

[An91] Andrews, G., *Concurrent Programming: Principles and Practice*, Benjamin-Cummings, 1991.

[At90] Atwood, T., "Two Approaches to Adding Persistence to C++," *Proceedings of the 4th International Workshop in Persistent Object System,* 1990.

[Ba*91] Balter, R., J. Bernadat, D. Decouchant, et al, "Architecture and Implementation of Guide, an Object Oriented Distributed System," *Computing Systems*, Vol. 4 No. 1, Winter 1991.

[Br*89] Bretl, R. , et al., "The Gemstone Data Management System," *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky eds., Addison-Wesley Publishing Co. 1989.

[Ca*86] Carey, M., D. Dewitt, D. Frank, G. Graefe, J. Richardson, E. Shekita and M. Muralikrishna, "The Architecture of The EXODUS Extensible DBMS," *Proceedings of the 12th International Workshop on Object-Oriented Database Systems*, September 1986.

[Ca*89] Carey, M. J., D. J. DeWitt, J.E. Richardson, and E. J. Shekita, " Storage Management for Objects in EXODUS," Object Oriented Concepts, *Databases and Applications*, Frontier Series, Addison Wesley, 1989.

[Co86] Cox, B., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[Co*84] Cockshot, W. , M. Atkinson, et al, "Persistent Object Management System," *Software Practice and Experience*, January 1984

[Da*91] Dasgupta, P., R. J. LeBlanc, M. Ahmad and U. Ramachandran, "The Clouds Distributed Operating System," *IEEE Computer*, November 1991.

[ElNa89] Elmasri, R. and S. Navathe, *Fundamentals of Database Systems,* Benjamin-Cummings, CA, 1989.

[Fo*88] Forin, A., J. Barrera, M. Young and R. Rashid, "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach," Carnegie-Mellon University Technical Report, CMU-CS-88-165, Aug. 1988.

[Ford*88] Ford, et al., "Zeitgeist: Database Support for Object-Oriented Programming," *Proceedings of 2nd International Workshop on Object-oriented Database Systems*, 1988.

[HeJo91] Hennessy, J. and N. Jouppi, "Computer Technology and Architecture: An Evolving Interaction," *IEEE Computer*, September 1991.

[HePa90] Hennessy, J. and D. Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann, 1990.

[HoMo90] Hoskins, A. L. and J. E. B. Moss, "Towards Compile-Time Optimizations for Persistence," *Fourth International Workshop on Persistent Object Systems*, Sept. 1990.

[HoZd87] Hornick, M. and S. Zdonik, "A Shared Segmented Memory System for an Object-Oriented Database," *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, 1987.

[KhCo86] Khoshafian, S. and G. Copeland, "Object Identity," *Proceedings of Object-oriented Programming System, Language and Application*, 1986.

[Kim*88] Kim, W., N. Ballou, H. T. Chou, J. Garza and D. Woelk, "Integrating an Object-Oriented Programming System with a Database System," *Proceedings of the OOPSLA-88 Conference*, September 1988.

[Kim90] Kim, W., "Architectural Issues in Object-Oriented Databases," *Journal of Object-Oriented Programming*, March/April 1990.

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

[LiHu89] Li, K. and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, 7:4, pp 321-359, November 1989.

[Ma*86] Maier, D. , J. Stein, A. Otis and A. Purdy, "Development of an Object-Oriented Database," *Proceedings OOPSLA-86*, November 1986.

[MaGu89] Marques, J. A. and P. Guedes, "Extending the Operating System Support for an Object-Oriented Environment," *Proceedings OOPSLA-89*, October 1989.

[Me92] Meyer, Bertrand, *Eiffel: The Language*, prentice Hall, New York, 1992.

[MeLa87] Merrow, T. and J. Laursen, " A Pragmatic System for Shared Persistent Objects," *Proceedings of the OOPSLA-87 Conference*, October 1987.

[Mo90] Moss, J. E. B., "Design of the Mneme Persistent Object Store, " *ACM Transactions on Information Systems*, Vol.8, No. 2, April 1990.

[NiLo91] Nitzberg, B. and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, August 1991.

[PeSt87] Penney, D. J. and J. Stein, "Class Modifications in the GemStone Object-Oriented DBMS," *Proceedings of the OOPSLA-87 Conference*, October 1987.

[Ra*87] Rashid, R. J., A. Tevanian, M. Young, D. Golub, R. J. Baron, D. Black, W. J. Bolosky and J. Chew, "Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[Ri90] Richardson, J., "Compiled Item Faulting: A New Technique for Managing I/O in a Persistent Language," *Fourth International Workshop on Persistent Object Systems,* Sept. 1990.

[Ro90] Rosenberg, J., "The MONADS Architecture: A Layered View," *Fourth International Workshop on Persistent Object Systems*, September 1990.

[Sc*86] Schantz, R. E., R. H. Thomas and G. Bono, "The Architecture of the Cronus Distributed Operating System," *Proceedings of Sixth International Symposium on Distributed Computer Systems*, May 1986.

[ShZw90] Shekita, E., and M. Zwilling, "Cricket: A Mapped Persistent Object Store," *Fourth International Workshop on Persistent Object Systems,* September 1990.

[Sk*86] Skarra, A. , S. Zdonik and S. Reiss, "An Object Server for an Object-Oriented Database System," *Proceedings of Workshop on Object-Oriented Databases*, 1986.

[Sr*90] Sriram, D. , R. Logcher, et al., "A Case Study in Computer-Aided Cooperative Productive Development," IESL Technical Report IESL-90-01, 1990.

[Sr*91] Sriram, D. , et al., "COSMOS: an Object-Oriented Expert System Shell," Technical Report, IESL, MIT, 1991.

[St88] Stroustrup, B., "Parameterized Types for C++," *Proceedings of the 1988 Unix C++ Conference*, 1988.

[Va*92] Vaughan, F. , et al., "Casper: a Cached Architecture Supporting Persistence," *Computing Systems*, Vol. 5, No. 3, Summer 1992.

[Ve92] *Versant Object Technology*, C++/VERSANT Reference Manual, Version 1.7, June 1992.

[Ve*90] Velez, F., V. Darnis, D. DeWitt, P. Futtersack, G. Harris, D. Maier and M. Raoux, "Implementing the O^2 Object Manager: Some Lessons," *Proceedings of the Fourth International Workshop on Persistent Object Systems*, September 1990.

[Wi90] Wilson, P, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Space on Standard Hardware," Technical Report, Electrical Eng. and Computer Science Department, University of Illinois at Chicago, 1990.

[Yo*87] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron, "The Duality of Memory and Communication in the Implementation of a Multi-processor Operating System," Proceedings of the 11th Symposium on Operating Systems Principles, Nov. 1987

[Yo89] Young, Michael W., "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System," Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, CMU-CS-89-202, November 1989.