1 Project Identification Information

Project Title: Mach Shared Objects

Award Dates: from July 1, 1991 to December 31, 1994

Institution and Address: University of Utah
302 Park Building, Salt Lake City, Utah 84112

Award Number: Arpa order 8223; R&T 4331818—01
Office of Naval Research Grant N00014-91-J-4046

Program Manager: David Gunning

Program Name: Persistent Object Bases
From: Director, Office of Naval Research, Seattle Regional Office, 1107 NE 45th St., Suite 350, Seattle, WA 98105
Subj: RETURNED GRANTEE/CONTRACTOR TECHNICAL REPORTS

1. This confirms our conversations of 27 Feb 97 and 11 Jul 97. Enclosed are a number of technical reports which were returned to our agency for lack of clear distribution availability statement. This confirms that all reports are unclassified and are “APPROVED FOR PUBLIC RELEASE” with no restrictions.

2. Please contact me if you require additional information. My e-mail is silverr@onr.navy.mil and my phone is (206) 625-3196.

ROBERT J. SILVERMAN
2 Summary

2.1 General Overview

The commonality of Unix-based open systems has greatly advanced software portability, sharing and interoperability. The MSO project exploited that commonality in constructing two object managers (OM) layered on modern operating systems: one for C++ and one for Common Lisp programs. Both incorporate comprehensive support for persistence of all language object types, data evolution, and distributed implementation and access.

Key to the MSO approach is viewing object management as an instance of the generalized problem of accessing and combining software components. Under this view, software components as managed as pervasive system resources rather than transient language artifacts. Hence the MSO OM focuses on efficient manipulation and sharing of modules as computational resources, including system and user libraries, data schemata, class implementations, and persistent objects.

This has been achieved through OMOS (Object/Meta Object Server), which provides comprehensive module definition, combination and mapping services. These capabilities are exploitable in O-O languages through abstractions for object naming, data type management, persistence, modularity, distribution, and shared object mutation control. Compiler and platform heterogeneity are obtained through system-generated objects describing module types.

Effective utilization in Alpha_1, a large-scale computer aided geometric design (CAGD) and manufacturing (CAM) software system, constituted our primary validation criterion. This was a demanding test case, due to its code size, versatility, and existing hand-crafted methods for persistence and data evolution.

- External publications: [LK92]

2.2 Persistent C++

Two software prototypes were constructed providing implementations of persistent objects for C++. The run-time type information necessary for fully polymorphic manipulation of C++ objects is generated in the original prototype by a modified compiler, and in the follow-on prototype by a compiler independent preprocessor. In both systems, this run-time type information is used by polymorphic load and store functions for persistent objects.

The Alpha_1 project has replaced their Lisp-based modeling language with one based on a reuse-oriented C++ library. Central to this task is building standardized language interfaces to their geometric C++ library. Writing these interfaces is tedious and thus error prone. To avoid this problem the Alpha_1 team is experimentally incorporating goofie, the MSO dossier generator, to systematically build suitable interfaces to the geometric objects in the Alpha_1 C++ library. Since the library contains over 1,000 distinct class declarations an automated tool for interface generation is expected to dramatically reduce the effort for this project.

- External publications: [MCLY94]
2.3 Persistent Lisp

A persistent Lisp prototype (UCL+P) was developed. It fully supports Common Lisp, and provides persistence facilities for all Common Lisp data types — not just objects — via an atomic transaction facility. In addition to Lisp runtime system modifications, the UCL+P implementation relies compiler level changes to Utah Common Lisp (UCL) to efficiently and transparently manage the loading and storage of Lisp data items. UCL+P confers persistence via reachability; when a volatile entity is accessible from a persistent entity, it becomes persistent.

The changes centered around a new implementation of the Lisp reference model; instead of using direct pointers to refer to Lisp items, a double pointer scheme was used. The intermediate pointers are grouped into a table called the Indirection Vector (IV). The IV has room for access marking bits used to track read and write access to heap resident entities. The IV also allows easy relocation of newly persistent entities into a protected region so that UCL+P may guarantee transaction semantics. Tests show that when processing nonpersistent data, the prototype runs about 25% slower than the original Lisp system.

2.4 Module Manipulation

A system is said to be compositionally modular if it facilitates a high degree of reuse of a simple notion of modules (namespaces) via a set of mechanisms that support effective decomposition and recomposition of programs. Etyma is an object-oriented application framework for compositionally modular systems, realized in the C++ language. The framework enables one to easily and rapidly build language processors for compositionally modular systems. An instantiation of Etyma classes essentially constitutes the internal representation of a source program in such a language. Etyma has been directly reused to build three completions: an interpreter for a module extension to the Scheme programming language, a compiler front-end for a compositional Interface Definition Language, and a compositional document processing system based on LaTeX. In addition, concepts developed in conjunction with Etyma have been reused in the specification of a layered architecture for composing object file components to construct entire applications. A description of this architecture has been awarded “Best Paper” at the 1995 International Workshop on Object-Orientation in Operating Systems.

1Art Lee was funded by sources other than the Mach Shared Objects project.
OMOS is an active object server which manages executable programs and their components as a set of composable objects. Besides replacing the traditional services of a linker, OMOS provides dynamic object loading, shared libraries, program analysis, and custom application construction.

A shared library implementation was prototyped using the OMOS class server under the OSF/1 and HP/UX operating systems. OMOS constructs classes using descriptions that are stored in containers known as meta-objects. Meta-objects live within a hierarchical namespace which is exported to clients. Clients use the namespace to identify classes they wish to instantiate. The meta-object acts as a level of indirection between the client and the class implementation.

OMOS caches the implementations it generates. As a result, multiple clients instantiating the same class share the same (physical) implementation of that class. When OMOS is used as the system loader, applications that are constructed from several common pieces (libraries) automatically gain the memory sharing benefits associated with shared libraries. Since program instantiations are made through a level of indirection (provided by meta-objects), library implementations can be specialized to take advantage of the common address space layouts. Through this specialization, use of highly general and often inefficient position independent code is not required, and expensive dynamic symbol name resolution can be avoided.

The OMOS shared library implementation was ported to both microkernel and macrokernel Unix systems on the PA-RISC and Intel 386 platforms. Average efficiency improvements of 20% speedup in execution time result for short running programs invoked using OMOS.

As stated above, CAGD/CAM was the driving application for the MSO approach to object management. It was soon realized, however, MSO's generalized approach applied equally well to management and manipulation of software components as objects. In particular, the OMOS technology constitutes a foundation for the development of a nanokernel-based decomposed operating system that achieves high performance while retaining inter-component protection and rich functionality. Such a system would overcome the performance/protection and performance/functionality tradeoffs that thwart traditional microkernel-based operating systems. This objective includes integration of selected research results of others, and free distribution of an unencumbered and usable version of the entire system.
This led to the formation of the the Flux project, with funding from ARPA/CSTO (contract DABT63-94-C-0058). Flux is developing an operating system that provides a much higher degree of flexibility than do traditional systems, and uses that added flexibility to circumvent the performance/functionality tradeoffs that thwart traditional highly-decomposed, microkernel-based operating systems. Foundations for this work include many MSO project themes, including (i) a comprehensive notion of modules, (ii) module manipulation cast as a system service, and (iii) a semantically enriched notion of module compatibility and adaptability. Items (i) and (ii) directly arose from the MSO project, while (iii) is a new emphasis motivated by pragmatically important inter-module concerns such as address space sharing, storage management policies, and levels of trust.

- External publications: [LHFL93] [FL93] [CFH+93] [FL94] [Orr95]

2.7 Web Page

www.cs.utah.edu/projects/mso
References


Dear Sir or Madam:

Enclosed please find the final technical report for grant N00014-91-J-4046 “Mach Shared Objects”.

I apologize for the delay in filing this required report.

Sincerely yours,

Gary Lindstrom
Professor of Computer Science

cc: Ronald P. Moody
Phone: 801-581-5586
E-mail: Lindstrom@CS.Utah.EDU


