Fast and Flexible Shared Libraries

Douglas B. Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg

University of Utah

Abstract

Existing implementations of shared libraries sacrifice speed (in loading, linking, and executed code), for essential flexibility (in symbol binding, address space use, and interface evolution). Modern operating systems provide the primitives needed to make the dynamic linker and loader a persistent *server* which lives across program invocations. This can provide speed without sacrificing flexibility. The speed is gained primarily through caching of previous work, i.e., bound and relocated executable images and libraries. The flexibility comes from the server's being an active entity, capable of adapting to changing conditions, modifying its cached state, and responding to user directives. In this paper we present a shared library implementation based on OMOS, an Object/Meta-Object Server, which provides program linking and loading facilities as a special case of generic object instantiation. We discuss the architecture of OMOS and its support of module binding primitives, which make it more flexible and powerful than existing shared library schemes. Since our design does not require any support from the compiler, it is also language-independent and highly portable. Initial performance results, on two operating systems, show an average speedup of 20% (range 0 – 56%), on short running programs.[1]

1 Introduction

The typical implementations of shared libraries hold different places on the static—dynamic spectrum. On one extreme we find the original System V shared libraries which need all resources to be fixed at link time — including the location of the libraries. On the other extreme we find shared library implementations whose symbols are resolved dynamically at run time.

Clearly, the former performs well because there is little work that must be done at run time — but there is little flexibility in this approach. Dynamic solutions are usually slower because there is more work to be done. But it is often the case that the work done is a repetition of work already performed — if the same memory image is produced 20 times, the work done for the last 19 is unnecessarily repeated in order to retain flexibility.

If we approach the object and executable files as the only possible sources of executable instructions on a system, our choices are limited. These files are static entities with limited semantics, constraining their flexibility. In principle, the only version of a program that matters is the image executing in memory (the final implementation). If we view the object file merely as a convenient intermediate form, our options become more interesting.

We describe a scheme which provides the flexibility of dynamic approaches while retaining the speed of static designs, and in some cases, improving on it. This design is based on OMOS, a server which is capable of dynamically generating executable images. As a server, OMOS is capable of reacting to changing conditions and maintaining the flexibility of traditional shared libraries, without requiring special compiler

[1]This research was sponsored in part by the Hewlett-Packard Research Grants Program and by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014–91–J–4046. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of ARPA, the U.S. Government, or Hewlett-Packard.

support. OMOS treats executable images as a cache, translating from more expressive forms (e.g., `.o`'s, or source modules) as necessary. By treating executables as a cache, OMOS avoids unnecessary repetition of work. As a natural side effect of its cache management, OMOS provides the sharing found in shared library systems.

2 Shared Libraries

2.1 Benefits and Caveats

The traditional benefits of shared libraries are well known. Their original primary motivation derived from their *shared* aspect, which reduced main memory use. This was the driving motivation some years ago, when memory was more of a scarce resource. Sharing also reduces disk consumption and eases software maintenance, since a library fix is instantly incorporated into all clients of that library. As shared library implementations became more sophisticated with support for *dynamic binding*, this allowed their exploitation for environment configuration. For example, internationalization is often supported through an environment variable.

However, shared libraries have other important benefits which are not as widely recognized. An apparently mundane one, but very important in a development environment, is drastically reduced static linking time. Statically linking a multi-megabyte binary, not at all unusual with the huge libraries now existing, can take many minutes. This becomes at least a factor of three worse when writing to a traditional NFS implementation, in which writes are synchronous. The lazy procedure binding used by most current shared library schemes, makes acceptable the run time performance of shared library versions of such programs. However, the majority of the savings probably comes from avoiding the I/O in writing out huge binaries.

Another less frequently recognized and exploited benefit, but of great importance and power, is the flexible binding afforded by modern shared library implementations. When procedure binding is dynamic, one can, in principle, *substitute* a different procedure at run time, or interpose a *wrapper* procedure around the original. This provides powerful flexibility, separating interface from implementation. This capability has been noted before[7], as well as the similarity between objects and shared libraries[18].

However, existing implementation of shared libraries have significant problems. They offer only limited control over the flexible binding discussed above. The working set sizes of programs increase, because the library functions an application uses are scattered across a large set of pages with other, unused functions. Memory savings can be minimal, at least with with small libraries, due to the size of the dispatch tables[11]. In general, the memory savings from shared libraries are probably more significant in a multi-user time-shared system than in the dedicated workstation environment common today, though there are no good measurements of this.

2.2 Issues

The traditional issues in shared library design and implementation are well recognized. For a complete discussion, refer to the articles by Sabatella[15] and Gingell[8], who give good overviews. These include granularity of sharing, granularity of symbol binding, code purity and the methods to achieve it, address binding, time of binding, time of loading, and version control. Additional issues have since surfaced, including handling languages such as C++ which present special problems such as static initializers[16].

3 Overview of OMOS

3.1 Overview

The OMOS object/meta-object server is a process which manages a database of *objects* and *meta-objects*. Objects are code fragments, data fragments, or complete programs. These programs may embody familiar services such as `ls` or `emacs`, or they may be functions or objects providing library-like services such as a binary tree object. Meta-objects are templates describing the construction and characteristics of objects, and contain a class description of their target objects.

OMOS uses the meta-object to construct and load implementations of a given class into client address spaces. For example, given a meta-object for `ls`, OMOS can create an instance of the `ls` class for a client. Instantiating an object subsumes linking and loading a program in a more traditional environment. By caching intermediate results of the construction process, OMOS can avoid unnecessarily reproducing effort when instantiating program binaries (e.g., parsing executable file formats, setting up mappers, etc.). When several applications within OMOS share sub-objects (such as those that make up conventional libraries), the effect is to share the underlying physical memory. In this way, by providing a shared object service, OMOS indirectly provides a shared library service.

3.2 OMOS and Class Instantiation

Client programs communicate with OMOS via an IPC mechanism, requesting instantiation of classes via a meta-object specification. OMOS maintains and exports a hierarchical namespace, whose names represent meta-objects, executable code fragments, or directories of other objects. Typically, clients request instantiation of meta-objects by name. An instantiation request may be accompanied by a *specialization*, indicating specific qualities the resultant implementation is to possess (including adherence to a given address space layout, required program facilities, etc.). If OMOS has a cached image associated with that meta-object that meets the user's specifications, the image is returned. If no cached image exists, OMOS constructs one by following the plan encoded within the meta-object. Ultimately, the cached image is mapped into the user's address space.

Meta-objects contain a specification, known as a *blueprint*, which describes how to combine objects and other meta-objects to produce an instance of the class. These rules map into a graph of operations, the *m-graph*. To generate an executable image, OMOS parses the blueprint and constructs an m-graph. The m-graph is executable; execution of the m-graph will generate an implementation of the class. Before executing the m-graph, OMOS applies any user-specified specializations to it, transforming the m-graph as appropriate.

Execution of the m-graph may result in OMOS compiling source code, performing symbol translations, and combining and relocating fragments. An m-graph operation may have as its operand other m-graph operations; the operation may in turn expand and execute its arguments. After successful execution of the m-graph, the resultant mappable image is cached and returned to be mapped into the user's address space.

3.3 OMOS Blueprints and the *Jigsaw* Operators

Since one of OMOS' primary functions is to link and load programs, an important focus of the blueprint operations is manipulation of symbol namespaces. A subset of the graph operations comprise *module operations*, as defined by Bracha and Lindstrom in the language *Jigsaw*[3, 4]. Jigsaw was designed in in order to *decompose* the bundled module definition and manipulation found in programming languages, including the manipulation performed in *inheritance*.

Conceptually, a module is a self-referential naming scope. Module operations operate on and modify the symbol bindings in modules. The modified bindings define the inheritance relationships between the component objects. The Jigsaw operators can be naturally composed to perform function interposition and inheritance as well as overriding or renaming of functions or data. The set of graph operations into which a blueprint may be translated is described in more detail in Section 3.3.

An advantage of representing the object construction as an executable graph is that it lends itself naturally to program transformations. A given operation may transform all or part of any of its operands. The specialization phase of evaluation may transform an entire meta-object. The sorts of transformations OMOS is capable of performing range from transparent interposition of monitoring routines to constraining objects to be bound within certain address ranges. Since OMOS is an active entity, the types of transformations it performs may change based on feedback or changes in the execution environment (as in the case of program monitoring where the execution of the program changes the implementation OMOS generates).

Currently, the specification language used by OMOS has a simple Lisp-like syntax. The first word in an expression is a graph operation (described below) followed by a series of arguments. Arguments can be the names of server objects, strings, or other graph operations. Each operation produces a module as its output. In this example

```
(merge
   /lib/crt0.o /obj/ls.o
   /lib/libc)
```

the `ls` program is constructed by merging (linking) two fragments with another meta-object, which represents the Unix library `libc`.

As indicated, m-graphs are composed of nodes which may contain graph meta-objects, or fragments. The complete set of graph operators defined in OMOS is described in [13]. The graph operators important to this discussion include:

Merge: binds the symbol definitions found in one operand to the references found in another. Multiple definitions of a symbol constitutes an error.

Override: merges two operands, resolving conflicting bindings (multiple definitions) in favor of the second operand.

Freeze: makes permanent a given symbol binding.

Restrict: virtualizes a set of bindings: any existing bindings become unbound and any existing definition for the symbol are removed.

Project: is the complement of restrict. project virtualizes all but a given set of bindings.

Copy-as: duplicates the value of a symbol definition under a new name.

Hide: removes a given set of symbol definitions from the operand symbol table, freezeing any internal references to the symbol in the process.

Show: is the complement of hide which hides all but a a given set of symbol definitions.

Rename: systematically changes names in the operand symbol table. Names may be references, definitions, or both.

Initializers: generates C++ static initializers for the C++ objects found in the file.

Specialize: specializes the operand to behave in a particular fashion (e.g., as either a fixed-address or dynamically loaded shared library, etc.)

Source: produces a fragment from a **C**, **C++**, or assembly language source object.

Constrain: directs OMOS to try to use or avoid a given address region when loading an object. The result of this operation can be mapped into a user address space.

List: groups two or more server objects into a list.

The leaf operands to OMOS operations are relocatable object files. OMOS manipulates relocatable object files using an idealized interface for symbol manipulation. High-level OMOS operations eventually translate into operations on an object file symbol space. OMOS provides a facility that allows many different name configurations ("views") to be mapped onto a given object file, allowing fast, efficient, incremental modification of a symbol namespace. Module operations typically take a regular expression as a specification of the symbols to select. Execution of a module operation (with the exceptions of merge and freeze) results in the production of a new view of the operand.

3.4 OMOS and Specialization

A powerful aspect of the OMOS framework is the use of executable graphs to represent the operations to be performed when constructing class implementations. Specialization adds another level of flexibility to the use of meta-objects. A base meta-object, representing an idealized class specification, can be transformed in various ways by OMOS. Using specialization, OMOS translates the underlying m-graph into a new, specialized m-graph.

For example, OMOS uses specialization to choose the style of shared library to generate from a given meta-object. As described in Section 4, OMOS supports two basic styles of shared libraries, with very different invocation mechanisms. The base implementation and the abstraction of the library is the same in both cases. To clients of the different schemes, only the invocation mechanism differs.

OMOS manages these differences as cases of specialization of a base library implementation. The user links through a special meta-object that transforms the target library into the appropriate form. The concept of the idealized library is maintained. For example, the operation:

```
(specialize "lib-dynamic" /lib/libc)
```

produces a library with dynamic routine resolution, while the operation:

```
(specialize "lib-constrained" (list "T" 0x1000000) /lib/libc)
```

produces a fixed-address version of `libc` whose text segment is constrained to live as close to the address 0x1000000 as possible. Merging with the first operation would result in the client's being linked with a set of stubs that execute at run time. Merging with the second operand would result in a client that is bound directly to an implementation of `libc` chosen by OMOS' constraint system. Both instances represent the notion of merging a client with an abstract "libc," although the specific implementations are very different.

3.5 OMOS and Constraints

OMOS uses its specialization facility to maintain flexibility in its shared library implementation. The address constraint specialization is a simple, yet powerful mechanism for guiding the placement of libraries within an address space.

OMOS describes an address space in terms of prioritized constraints. A required constraint is that no two objects may overlap. A highly desired constraint is that existing implementations be reused. Other weaker constraints, optionally provided by the user, may specify desired placement of the object (e.g., library) within the address space. When no existing implementation meets all the given constraints, OMOS will generate (and cache) a new one. If an existing implementation meets the given constraints, OMOS will reuse it, causing its read-only portions to be shared among its different clients.

In the case that a user attempts to make use of an address space region that is in conflict with existing libraries, the constraint system considers the priorities associated with each requirement and resolves the problem by using alternate implementations or generating new ones. Subsequent invocations of the same combination of applications and libraries will use the existing set of implementations.

4 OMOS Shared Library Schemes

As previously mentioned, OMOS supports different strategies for implementing shared libraries. A *library* class, derived from the standard meta-object class, is used to describe OMOS shared libraries. The class specifies a default specialization to be applied to the meta-object which may be used to generate different implementations. In figure 1 we see a sample implementation of the library "libc". The first line specifies the default specialization to be applied to the implementation (producing a self-contained shared library in this example). The rest of the meta-object describes how to construct the base implementation of the library.

4.1 Self-contained Shared Libraries

In the standard scheme for implementing shared libraries in OMOS, OMOS maintains both the client application and library as objects internal to itself. OMOS makes use of its constraint system to allow

```
(constraint-list "T" 0x100000 "D" 0x40200000)  ; default address constraint
(merge
  /libc/gen /libc/stdio /libc/string /libc/stdlib
  /libc/hppa /libc/net /libc/quad /libc/rpc)
```

Figure 1: Sample Library Meta-Object

different address space configurations to be used, without explicit intervention of the user or a system manager.

In this scheme, each implementation of an application that references a library will be bound to a particular version of the library. The library version will be located at a fixed address. As a result, all resolution of addresses and linking is done at the time a fixed version of the application is constructed. Typically, in a development environment, fixed versions of clients and libraries are generated at installation time, to avoid the startup latency on first invocation.

If the application meta-object specifies a library for which there is no implementation, or for which the existing implementations conflict with other address space requirements, OMOS will use its constraint system to resolve the conflict. OMOS' constraint system will cause new versions of some number of libraries to be generated in a configuration in which there are no conflicts. In the common case only one implementation of each library will ever be generated. Currently, a little planning by the system manager helps optimize this. However, OMOS could easily record the conflicts found, and occasionally the system manager could feed that data into OMOS' constraint system to determine better placements, or this could be done fully automatically. (It should be noted that it is fairly important that few versions of libraries be generated, since disk space for caching multiple versions of large libraries could be significant.)

In this way, OMOS achieves the same flexibility as dynamic shared library schemes based on position independent code (PIC), as well as the traditional sharing benefits, without the run time overheads of dynamic binding. By using its active nature, OMOS can specialize its implementations to the actual address requirements of its clients, rather than relying on the availability of a completely general framework such as PIC.

As is normal with shared libraries not using PIC, variables shared between application and libraries, or between two libraries, can pose problems in this scheme. If shared variables are defined in the client application, and referenced by the library, the physical sharing of the library by two different applications can not be maintained. References may exist in the client application, but, in general, all definitions of variables must be made in the library "furthest downstream" that references them. No circular references may exist. Similarly, if libraries directly reference user procedures, OMOS will have to construct a new library image for each different application. OMOS or the system manager can monitor such occurrences, and if they are frequent, specialize the offending libraries to dispatch via a branch table.

The use of self-contained shared libraries poses inconveniences for debugging, since the program initially executed by the user may be a bootstrap loader whose only function is to load the client application and libraries. As a result, by default, the application debugging symbols will not be present in the file. With a little extra trouble (e.g., use of the `gdb` "symbol-file" command), the client symbols can be loaded in after the client and libraries have been mapped in. As a more elegant solution, we plan to enhance `gdb` to interface directly with OMOS, allowing them to interact seamlessly.

Self-contained Shared Library Speed and Memory Use

The self-contained shared library scheme benefits from simplicity. Linking work done to resolve references to library routines need not be repeated in order to preserve flexibility. In the common case, libraries can reside at a single location. Thus, this strategy tends to optimize the common case. The amount of work required to load a cached executable is constant, where schemes that do dynamic link resolution to maintain flexibility often must do work in proportion to the number of external references made by the client, every time the library is loaded.

Since the self-contained shared libraries have no dispatch table, the absolute memory requirement for applications is decreased. For small programs, shared library dispatch tables can account for a surprising portion of program size. (Initial measurements of the SunOS implementation have shown that for small programs (e.g. `ls`) and libraries (`libc`), more memory is used for dispatch tables than is saved in library code[11].)

In addition, the self-contained shared library scheme can use absolute addressing modes to reference data, which has particular performance benefits, important on CISC machines but also relevant to RISC architectures. Use of the OMOS constraint system does not preclude the use of position-independent code, of course. In fact, the availability of PIC makes resolution of address constraints trivial, since there is effectively no cost to relocate an implementation. PIC is not required, however, to use the base functionality, which renders this scheme simple and portable.

OMOS also benefits from special optimizations that can be performed due to its being an active entity. One such optimization is reordering code based on function usage in order to improve locality of reference. OMOS can automatically generate implementations that will produce monitoring data, which it will then use to derive a preferred routine order. This reordering benefits both cache performance and paging behavior. We have performed this experiment and achieved average speedups in excess of 10%, as described in [14].

4.2 Partial-image Shared Libraries

An alternative scheme for shared libraries is the *partial-image* shared library, which provides more convenient application debugging facilities. In partial-image shared libraries, the client program consists of a complete copy of the application code, which is exported to the user (normally, OMOS manages all executable images as a cache and does not expose individual executables to the user).

The partial-image application contains stub routines for each library entry point. On the first invocation of a routine in a library, the client stub contacts OMOS and loads in the library, returning the address of a hash table containing the addresses of all library routines. The first time a function in a dynamically loaded library is accessed, its name is looked up in the function hash table and the value of its entry point is stored in an indirect branch table. Subsequent invocations of the function are made through the pointer in that table.

The specialize module operation provides a framework for implementing dynamically linked shared libraries within OMOS, using two types of specialization. The "lib-dynamic" specialization generates an m-graph which implements the application's access to the library (i.e., the stubs which perform dynamic loading), and the "lib-dynamic-impl" specialization generates the m-graph which will produce the library implementation that is to be loaded and shared.

The "lib-dynamic" specialization creates an m-graph, the evaluation of which causes stub functions to be dynamically generated for each referenced entry point in the operand. The stub code is compiled and returned as the representative implementation of the library. On future invocations of the specialization the cached copy of the compiled function stubs will be used. The function stubs also contain a reference to the "lib-dynamic-impl" specialization of the library. On execution of the client application, the "lib-dynamic-impl" implementation of the library is loaded from OMOS.

Partial-image shared libraries allow developers to make use of unmodified system debuggers to debug OMOS client applications. They also eliminate the need by the user for knowledge of the mapping process. In addition, partial-image shared libraries provide a more traditional interface to OMOS. Since a dynamically linked application is an executable file, the semantic meaning of file rename, copy, and delete are the same as with statically linked programs. In debugging environments, developers may prefer the higher degree of control provided.

The utility of partial-image shared libraries is limited by the use of shared variables. Since the application is fixed and outside of OMOS' control, references by the application to variables shared with libraries are problematic. Either the variables must live in a fixed segment whose address and size cannot change, or the application must use some level of indirection to access the shared variables. We could make use of shared variables within this context less inconvenient with compiler modifications, but one of our portability goals was to avoid reliance on particular compiler features. As a result, the practical utility of partial-image shared libraries will tend to be limited to debugging or to "well-behaved" libraries whose shared state is encapsulated and accessed via procedures. *Versioning* should be implemented and would provide safety, but it would not help with resource consumption, i.e., disk space.

5 Constructing and Executing Programs Using OMOS

All programs constructed in OMOS have a meta-object specification. Users of a shared library reference the library name symbolically as part of a "merge" or "override" operation within the client meta-object. For example, the meta-object seen earlier,

```
(merge
   /lib/crt0.o /obj/ls.o
   /lib/libc)
```

will resolve references from `/lib/crt0.o` and `/obj/ls.o` to symbols found in the default implementation of library `/lib/libc`. Execution of this set of operations will ultimately produce mappable, executable images representing the main program and the library `libc`. If we give this meta-object a name, the mappable result will be accessible to clients of OMOS.

There exist several different mechanisms for instantiating classes maintained by OMOS. One client program of OMOS is a small bootstrap loader used to load OMOS meta-objects. In Unix, we normally invoke this loader via the "interpreter" feature (`#! /bin/omos`). This allows us to export entries from the OMOS namespace into the Unix namespace, in a portable fashion (as a parameter in the file). When invoked, the bootstrap loader contacts OMOS via IPC, loads in the executable image(s) for a given meta-object, and jumps to its entry point, subsuming the functionality of the system call `exec()`[2]. In our example, if `/lib/libc` is a self-contained shared library, the bootstrap loader will map the main program in at one set of addresses, and the `libc` library at a different set. In the absence of address conflicts, all other clients of `/lib/libc` will share the same mapping and hence, the same physical memory as this client of `/lib/libc`. Another example of this mechanism is the partial-image scheme, in which applications contain the main program as well as code to contact the bootstrap loader.

However, use of the bootstrap loader involves a certain amount of overhead because the system must first load and execute the bootstrap before OMOS can do its work. Ideally, OMOS should be integrated into the operating system's implementation of the `exec()` system call, which we have done in the OSF/1 operating system running atop Mach 3.0. When an object in OMOS' exported namespace is requested to be executed, `exec` sets up an empty task and calls OMOS with handles to the task and the OMOS object. OMOS does its normal work, resulting in a set of mappable segments, which it then maps in to the target task. This replaces the portion of `exec` which is responsible for reading in object file contents. In general, OMOS should be able to do this more efficiently than the operating system, because it does not have to open files, parse complex object file headers, etc. Note that when OMOS is integrated in this manner, `/bin`, for example, can become a "filesystem" backed only by OMOS. The meta-objects and executable fragments providing the contents can be stored anywhere.

Program loading is a special case of the more general class generation and loading facility which OMOS provides. In addition to `exec()`-style invocation facilities, OMOS exports a more general interface for dynamically loading class implementations into executing programs. Via a meta-object, a client program specifies the class to be loaded, any specializations to apply to the meta-object, and a list of symbols whose bound values are to be returned from OMOS. The meta-object specification may either be the name of a meta-object found within the OMOS namespace, or an arbitrary blueprint to be executed by OMOS. This invocation mechanism is being used to load class methods in a project building a portable, persistent object management system on Unix[12].

[2] This is the invocation method we used to gather timings on HP-UX.

```
;;
;; malloc() -> malloc'()
;;
(hide "_REAL_malloc"
   (merge
      ;; Get rid of the old definition
      (restrict "^_malloc$"
         ;; stash a copy of _malloc() for later use
         (copy_as "^_malloc$" "_REAL_malloc"
            (merge /bin/ls.o /lib/libc.o)
         )
      )
      ;; Merge in a new definition
      /lib/test_malloc.o
   )
)
```

Figure 2: Interposition Example

As a note, execution of arbitrary blueprints can be used to implement a dynamic loading facility similar to that found in the dynamic linker, `dld`[9]. A client can request that new classes be loaded, which are then merged with its own implementation, allowing the new classes to refer to procedures and data structures within the client. A limitation of this scheme is that the client and target classes must both be within OMOS (not a problem in our view!) and the client must keep track of which classes it has dynamically loaded. We plan to add a facility to OMOS that will allow it automatically to maintain this information for interested clients.

Security is an obvious concern in the cases in which an unprivileged client invokes OMOS. In a system with a powerful IPC such as Mach's, this is not a problem. In normal Unix systems, the privileged port feature provides authentication that many utilities rely on— for OMOS to rely on it introduces no additional vulnerabilities. Thus the bootstrap loader can be **setuid**, providing an authenticated channel to OMOS. In general, the Unix file permission scheme can be used, applied to OMOS-managed objects.

6 OMOS Flexibility

The use of module operations and executable graphs by OMOS provides significant flexibility not found within conventional loading and linking environments. By using module operations grounded in theory, we provide complete symbol manipulation and binding facilities. The fact that OMOS is a server allows it to respond dynamically to program needs. That it is a server also makes it convenient to maintain program state across invocations.

Module operations can easily be used for interposing new routines within an executable. By invoking copy-as on all definitions of a given set of symbols using some well-known scheme (e.g., prepending a package name), then using restrict to virtualize the original bindings, new values for the symbols in question can be inserted transparently in the original application.

For example, in Figure 2, we produce a version of the C library, libc, where a new version of `malloc` has been inserted to trap calls to the original routine. References to the native routine in the new routine are preserved.

The source operator can be used to fill in missing variable or routine definitions with default values. The rename operation can be used, as seen in Figure 3, to do something more drastic, such as to rename all references to routines that should never be called to the routine `_abort`, which will produce notable behavior if called unintentionally.

```
(merge
    ;;  resolve an undefined data reference and
    ;;  reroute undefined routines to "abort()"
    (source "c" "int undef_var = 0;\n")
    (rename "^_undefined_routine$" "_abort"
        /lib/lib-with-problems))
```

Figure 3: Symbol Renaming and Resolution Example

As mentioned in Section 4.1, OMOS can transparently modify program executables to provide monitoring data, which can later be used to reorder the application to improve performance. OMOS does this by using module operations to extract the set of referenced routines and generate wrapper functions around each, to log entry and exit from the routine. The wrapper functions are interposed between each caller and the called routine.

In general, in performing the monitoring experiments and constructing shared libraries, we have been impressed with the utility of the module operations for manipulating symbolic interfaces. We have found ourselves using these operations to "tweak" object files in both simple and complex ways, where previously we would have edited a new copy of the C source file, or tediously generated and modified assembly code. We can see that link-level facilities such as this, while not terribly glamorous, are badly needed when combining objects produced in different contexts.

7 OMOS Portability

Because OMOS is implemented in a portable fashion and makes use of relatively unsophisticated operating system functionality, it is highly portable.

As its first major step towards portability, OMOS does not rely on the availability of PIC in order to provide a shared library service. As mentioned, OMOS can make use of PIC if it is available, but we contend that we provide the same flexibility as PIC at no additional cost in the common case (i.e., few redundant copies of cached libraries), with an improvement in speed in the common case.

OMOS is written in C++, and C++ objects have been used to encapsulate important operating system functionality. As a result, only a small number of highly localized changes must be made in order to port OMOS to a new system. OMOS requires support for some form of on-demand mapping from files (e.g., `mmap`), some file system I/O capabilities, and the ability to allocate heap memory at particular locations.

Optionally, invocation of OMOS can be integrated into the operating system `exec` code. This is straightforward in modern Unix systems. The only required extension to the Unix interface itself, is some way efficiently to map memory into an unrelated process, which we do on Mach with `vm_map()`. A simple and easy to implement extension to the BSD or System VR4 `mmap()` interface would provide this: `pmmap(pid, ...)`, a privileged call.

Finally, OMOS requires an understanding of the native object file format. Although this understanding has also been encapsulated in an object, it remains the most complex and messy portion of the system to port. A promising route for future portability is the GNU project's `BFD`[5] library. This library provides a machine and object-format independent interface to object files. It contains an array of object-format specific backends, and backends exist for most popular formats and machines. We are in the process of fitting this object file switch as a back-end to OMOS. Although a potential problem is inefficiency in speed and size, we expect this library will solve most of these portability issues.

Implications for Other Programs

There are a number of existing Unix utilities which read object files, such as debuggers, `nm`, `size`, and `strings`. Except for debuggers, each program is concerned with only a small part of the whole file, e.g., symbol table, string table, or `exec` header. They currently expect to see a simply structured sequence of bytes, from which they extract the few portions of interest. When such a program `open()`'s a file, it would make little sense for OMOS (integrated with the operating system, e.g. as a Virtual File System[10]) to pass it an entire byte stream, only to have the program discard the majority of the data. This is especially expensive and inappropriate when program segments are sparsely laid out in memory, which is beginning to occur on Unix systems.[3] In cases where OMOS is not integrated into the OS, the existing utilities would not comprehend the new formats presented.

A solution to all of these issues lies in the `BFD` library, which is already used by portable versions of these utilities. We are adding a new "OMOS" backend which directly invokes the server, requesting only those portions of interest.

8 Results

8.1 Status

OMOS is in experimental use as an object server running on top of the Mach operating system on PA-RISC (OSF/1 server) and ix86 (BSD server) platforms, and on HP-UX on the PA-RISC platform. OMOS supports communication via Mach IPC, Sun RPC, and System V messages. On HP's PA-RISC object file format, `SOM`, and on `a.out` format files, OMOS supports the Jigsaw module operators, as well as a number of other useful graph operations. Conversion of OMOS to use the BFD object file switch is underway. OMOS has been used to conduct experiments in automatically generating locality-of-reference optimization in running systems[14]. The basic OMOS system comprises 17,600 lines of C++ code, and uses C++ pointer overloading to implement an automatic reference counting scheme for memory management.

We also have a non-server version of OMOS, called the Object File Editor (OFE). It offers a traditional command interface and manipulates files in the normal Unix file namespace. OFE has proven very useful for manipulating object files in a traditional environment.

8.2 Performance

Methodology

All timings were collected on an HP9000/730 which had 64 MB RAM and two SCSI-2 disks (HP-UX), and 32 MB RAM and one SCSI-2 disk (OSF/1-MK). The HP730 has a 67 Mhz PA-RISC 1.1 processor with a pagesize of 4KB. On HP-UX, we used HP-UX version 9.01, AT&T's C++ 3.0.1 ("cfront"), and GCC 2.3.3.u3 (a Utah distribution of 2.3.3 with additional optimizations for the PA). On Mach (OSF/1-MK), we used our HP700 ports of the Mach 3.0 kernel, version NMK13, and the OSF/1 single server, version 1.0.4b1 (derived from OSF/1 1.0.4). All files were on local disk.

Timings were obtained using the GNU `time` program on HP-UX and with the `csh`'s built in `time` command on OSF/1-MK. (Note that on Mach, the "system" cpu time is currently meaningless (low), since separate threads in the server provide most system services to the user program.) The systems were in multiuser mode, but idle, as verified by `vmstat`. Each run was repeated at least three times, with very little variance observed, less than 2%.

Two programs were studied. One was Unix `ls` (the 4.3 BSD version on HP-UX and the OSF/1 version on OSF/1). The other was `codegen`, part of the Alpha_1 geometric modeling system, a large system totaling 650,000 lines of C++ and 150,000 lines of Lisp. `Codegen` itself consists of 5,240 lines of code stored in 32 files. In addition, it relies on six libraries: two Alpha_1 libraries as well as libm, libl, libC, and libc. An

[3] This is already a problem with existing shared library systems— one frequently gets spuriously huge core dumps due to such memory gaps.

Table 1: Constraint-based Shared Library Performance Times in Seconds

HP-UX

Test: `ls` (1000 iterations)	User Time	System Time	Elapsed Time	Ratio
HP-UX Shared Lib	4.16	2.23	16.45	
OMOS bootstrap exec	1.63	14.57	16.56	1.007

HP-UX

Test: `ls -laF` (1000 iterations)	User Time	System Time	Elapsed Time	Ratio
HP-UX Shared Lib	121.8	169.4	294	
OMOS bootstrap exec	80.8	189.2	276	.93

HP-UX

Test: `codegen` (1000 iterations)	User Time	System Time	Elapsed Time	Ratio
HP-UX Shared Lib	211.1	75.9	289	
OMOS bootstrap exec	182.4	54.5	238	.82

Mach 3.0 with OSF/1 Server

Test: `ls` (300 iterations)	User Time	System Time	Elapsed Time	Ratio
OSF/1 Shared Lib	.89	4.46	38	
OMOS bootstrap exec	1.50	5.62	23	.60
OMOS integrated exec	.89	4.49	17	.44

optimized, statically linked version of the program contains roughly 1,000 functions, with a total text size of 203KB and data size of 53KB. Tests were performed on a non-optimized, debuggable version with 289KB of text and 348KB of data. During timing the codegen program was run on a small input dataset which required reading three small files, and generated a single small file redirected to /dev/null.

Timings

In preliminary timing tests presented in Table 1, we compared the same implementation of `ls` constructed using HP-UX shared libraries[4], and OMOS self-contained shared libraries, configured to use System V messages to communicate with OMOS. For a simple `ls` performing a list of a directory with a single entry, the OMOS and HP shared libraries performed similarly. (HP-UX must do some relocations OMOS does not, but the OMOS bootstrap program must do some IPC that HP-UX does not.) When we increase the number of system calls (by adding the "-laF" flags to the `ls` command), OMOS performance becomes proportionally better, exceeding HP-UX's by 7%. This was due to "user" time increases in the HP-UX shared library case. Since we used the default `-B deferred` binding mode, the HP-UX implementation[6, 15] does lazy relocation on procedure, and to some degree, data references, this time was presumably spent doing relocations and dispatch table patching. In timing tests using the much larger `codegen`, OMOS shared libraries were 18% faster than HP-UX shared libraries. Again, we see the effect of increasing numbers of relocations performed on every program invocation in the case of of HP-UX, but only once for OMOS.

Both of the tested programs execute for relatively short times. On longer-running programs, the proportional speedup using OMOS would tend to be less, because in the traditional design, the majority of the

[4] All HP-UX timings are reported using demand-loaded binaries. Our tests showed that demand-loading made no significant difference to the results.

relocations are presumably performed at startup. However, our implementation is also paying a significant startup cost, the IPC from the bootstrap loader to OMOS. This would tend to counteract the above effect.

On Mach 3.0-OSF/1, we compared our implementation and the OSF/1 1.0 scheme[1]. Since different compilers[5] supported the two implementations of shared libraries, we could not ensure a completely valid shared library comparison, but we do see that the both the bootstrap and integrated `exec` versions of OMOS perform remarkably well, with the latter giving a 56% speedup. We are still in the process of measuring exactly where the gains are derived, but we believe that a shorter code path resulting from pre-parsing the executable file may be partially responsible. On tests made on the 386 version of Mach, OMOS integrated `exec` performed 33% faster than the native version, reinforcing this belief.

9 Related Work

A user-space loader is no longer unusual. Many operating systems, even those with monolithic kernels, now use an external process to do program loading involving shared libraries, and therefore linking. However, the loader/dynamic linker is typically instantiated anew for each program, making it too costly for it to support more general functionality such as in OMOS. Also, these loaders are not constructed in an extensible manner.

Other shared library designs give some degree of flexibility over symbol resolution. Functional substitution is usually supported, but without the fine control we offer— for example, it is difficult or impossible fully to control references from within shared libraries to a user-provided substituting or interposing function. It is common to support the runtime overriding of library search path rules by an environment variable (SunOS, System VR4, and HP-UX). Some versions of SunOS support the LD_PRELOAD environment variable which specifies the shared libraries to load before loading those requested by the application itself. In this way, some simple forms of runtime functional substitution can be provided. In addition, some linkers provide simple symbol renaming on the command line.

Packages exist, such as `dld`[9], to aid programmers in the dynamic loading of code and data. These packages tend to have a procedural point of view, provide lower-level functionality than OMOS, and do not offer the control over symbol manipulation that OMOS provides. `Dld` does offer dynamic unlinking of a module, which OMOS currently does not support. Since OMOS retains access to the symbol table and relocation information for loaded modules, unlinking support could be added. The CLAM system from the University of Wisconsin provides dynamic loading of C++ code in order to extend graphical user interfaces.

The Apollo DSEE[2] system was a server-based system which managed sources and objects, taking advantage of caching to avoid, typically, recompilation. DSEE was primarily a CASE tool and did not take part in the execution phase of program development.

10 Future Work

Many interesting problems remain to be addressed in this shared library scheme. We plan to develop policies whereby several instantiations of an OMOS meta-object, such as a shared library — each tuned for a different use — can be made available to client applications. Properly tuned policies would have beneficial effects on the working set sizes of programs using shared libraries.

The current constraint system uses primitive criteria for making decisions about object placement. A more sophisticated constraint system, based on the University of Washington's "Delta-Blue" constraint solver[17], has been developed in LISP and is being ported to OMOS and C++.

We are planning to extend the module operations to allow discrimination between symbol references and definitions. This will allow more flexible control over binding. For example, it will then be possible to better handle recursive functions.

There are many engineering issues to be addressed in OMOS: consolidating OMOS servers in a network,

[5] The implementation problem was the incompatible object formats each compiler generated, not the generated code itself.

refining the policies for managing main memory and backing store, and elaborating client interfaces. Using primitive fragments consisting of a single routine would gain flexibility, but it is unclear whether data structures can be made efficient enough for this to be feasible.

The extensible nature of OMOS, and its knowledge of everything from source file to execution traces, make it applicable to optimizations requiring run-time data. We are currently undertaking a project to exploit OMOS' specialization abilities to transparently optimize RPC. As another example, OMOS could transparently implement the type of monitoring done by MIPS' `pixie` system, to optimize branch prediction. Another direction is suggested by OMOS' natural connection with program development. OMOS could easily be used as the basis of a CASE tool, where its ability to feed back data from program execution would be useful for both debugging and optimization.

11 Conclusion

We have described a shared library design based on OMOS, a persistent server, providing fast and portable implementations of shared libraries, with superior flexibility both in symbol manipulation and implementation options. Preliminary performance results show it to be faster than some existing shared library schemes.

Acknowledgements

We are grateful to Peter Hoogenboom for providing OFE, BFD, and helping with other object format support. Thanks also go to Jeff Law and Mike Hibler for helping to integrate OMOS with OSF/1 `exec`, and to the other members of CSS, for providing their expertise in times of need.

References

[1] Larry W. Allen, Harminder G. Singh, Kevin G. Wallace, and Melanie B. Weaver. Program loading in OSF/1. In *Proceedings of the Winter 1991 USENIX Conference*, pages 145–160. USENIX Association, Winter 1991.

[2] Apollo Computer, Inc, Chelmsford, MA. *DOMAIN Software Engineering Environment (DSEE) Call Reference*, 1987.

[3] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992. 143 pp.

[4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20-23 1992. IEEE Computer Society.

[5] Steve Chamberlain. *The Binary File Descriptor Library.* Cygnus Support, Palo Alto, CA, 1992. in FSF `binutils` distribution; Copyright Free Software Foundation.

[6] Cary A. Coutant and Michelle A. Ruscetta. Shared libraries for HP-UX. *Hewlett-Packard Journal*, pages 46–53, June 1992.

[7] Robert A. Gingell. Evolution of the SunOS programming environment. Unpublished report, Sun Microsystems, Inc., 1988.

[8] Robert A. Gingell. Shared libraries. *Unix Review*, 7(8):56–66, August 1989.

[9] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software— Practice and Experience*, 21(4):375–390, April 1991.

[10] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Conference*, pages 238–247, Atlanta, GA, Summer 1986.

[11] John Kohl and Carol Paxson. How well do SunOS shared libraries work? Unpublished report, Computer Science Division, University of California at Berkeley, December 1991.

[12] Gary Lindstrom and Robert R. Kessler. Mach Shared Objects. In *Proceedings Software Technology Conference*, pages 279–280, Los Angeles, CA, April 1992. DARPA SISTO.

[13] Douglas B. Orr and Robert W. Mecklenburg. OMOS — an object server for program execution. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 200–209, Paris, France, September 1992. IEEE Computer Society.

[14] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993.

[15] Marc Sabatella. Issues in shared libraries design. In *Proceedings of the Summer 1990 USENIX Conference*, pages 11–24, Anaheim, CA, Summer 1990. USENIX.

[16] Marc Sabatella. Lazy evaluation of C++ static constructors. *ACM SIGPLAN Notices*, 27(6):29–36, June 1992.

[17] M. Sannella, B. Freeman-Benson, J. Mahoney, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. Department of Computer Science and Engineering TR-92-07-05, University of Washington, 1992.

[18] Donn Seeley. Shared libraries as objects. In *Proceedings of the Summer 1990 USENIX Conference*, pages 1–12, Anaheim, California, Summer 1990. Usenix Association.

Author Information

Douglas Orr is a Ph.D. student in the Computer Science Department at the University of Utah. His research interests include distributed systems, compilers and object systems. He is the primary architect of OMOS.

John Bonn is an M.S. student in Computer Science. His interests lie in operating systems and object oriented programming. Prior to joining the department John worked in industry for 5 years, specializing in real-time signal processing systems and Unix internals.

Jay Lepreau is Assistant Director of the Center for Software Science, a research group within Utah's Computer Science Department which works in many aspects of systems software. He has worked with Unix since 1979, and has served as co-chair of the 1984 USENIX conference and on numerous other USENIX program committees. His group has made significant contributions to the BSD and GNU software distributions. His current research interests include dynamic software system structuring for performance and flexibility, with operating system, language, linking, and runtime components.

Robert Mecklenburg is a Research Assistant Professor in the Computer Science Department. His major research interests are large scale software development, mixed language object-oriented programming, and persistence in object-oriented languages. He received his Ph.D. in Computer Science from the University of Utah in June, 1991. His dissertation, entitled *Towards a Language Independent Object System*, described a facility for mixing object-oriented programming languages with the goal of code reuse and new project development in multiple languages.

The author's addresses are: Center for Software Science, Department of Computer Science, University of Utah, 84112. They can be reached electronically at {dbo,bonn,lepreau,mecklen}@cs.utah.edu.

[0] HP-UX is trademark of Hewlett-Packard; Unix is a trademark of Unix System Laboratories.