Program Specialization Using the OMOS System

Douglas B. Orr Jay Lepreau Jeffrey Law

Department of Computer Science
University of Utah
Salt Lake City, UT 84112
{dbo,lepreau,law}@cs.utah.edu

Technical Report UUCS-95-016
March, 1995

Abstract

Abstraction and modularity provide many software engineering benefits. Hiding details of module internals can, however, prevent system implementors from being able to provide anything but a highly general implementation of a given module. We describe OMOS, a programmable linker/loader and system server that manages module implementations. OMOS allows system builders to describe system architectures in high-level terms, via a module construction scripting language. Using scripts, system implementors can provide modules that can test and react to both their static and run time environments. These modules, which we refer to as **electric libraries***, can produce implementations that are optimized at link or run time, without sacrificing modularity, expanding interfaces, or requiring changes in client programs. We identify and implement three types of specializations that OMOS can perform, and quantify the impact of two of them on a few standard Unix utilities: performance improvements ranged from 6% to 47%.*[1]

1 Introduction

In software engineering terms, a *module* is an independently developed unit of software with a well-defined visible interface and an opaque implementation. The uses of abstraction and modularity relieve programmers of the burden of understanding the details of the implementations of the software components they use. By restricting access to software modules through well-defined interfaces, many possible sources of programming error are eliminated. In modular software, the internal workings of the module are not accessible, so the implementor is free to change those specifics as needed without affecting its clients. The use of general, abstract notions in software permits development of flexible and reusable modules.

While the mechanisms of abstraction and modularity are a boon to software engineering, they also have some drawbacks. The same barriers that protect clients from relying on particular details of an implementation also constrain the effective use of that implementation. An interface that is too rich in detail is difficult to fully implement. In addition, clients of such an interface could be difficult to port to less rich environments. An interface low in detail is easier to implement, but provides less explicit information that can be used to guide selection of appropriate algorithms within its implementation.

[1]This research was supported in part by the Advanced Research Projects Agency under grant numbers DABT63–94–C–0058 and N00014–91–J–4046, and by the Hewlett-Packard Research Grants Program.

One approach to solve the above problems is *open implementations* — systems whose implementations are customizable via external means — is an active area of research[15, 7, 21]. Different approaches to customization have been taken. Customizing an open implementation can involve such diverse tradeoffs as whether to build implementation-specific information into clients or service providers, expand interfaces, or perform customizations statically or dynamically. Two important parameters are whether implementation information is couched in *imperative* or *declarative* form, and whether that information is *directive* or merely *informative*, i.e., whether the application is telling the system *what* to do to support its needs, or just *disclosing* what the application is doing[13]. Patterson and Gibson[21] make a strong argument that disclosure is superior in the operating system environment.

Our approach to solve the problems is OMOS, a fully programmable linker/loader and persistent server, that fills the roles taken by the linker, loader, shared library support, and `exec` operating system service in a traditionally structured system[20, 19, 18]. OMOS also provides the means to construct programs that are specialized to their expected operational conditions. This is done in two ways: by inferring operational characteristics from the symbolic names found in modules, or absent from them (implicit disclosure), and by explicit disclosure through annotations. OMOS provides a mechanism to annotate program components, and to examine and respond to those annotations. Through this mechanism OMOS provides a simple way to negotiate an appropriate implementation, without adulterating or complicating module interfaces.

OMOS modules use *disclosing* annotations. Rather than requesting specific support from the underlying system, a module makes general statements about its operating properties, in a declarative fashion. These statements can then be interpreted by service providers. In contrast, in a directive approach, a module that expects to do large amounts of I/O, for example, could express that it wishes to use virtual memory operations on buffers greater than a particular size. While that mechanism might be appropriate for one set of operating conditions, it could be completely inappropriate for another. So, instead, an OMOS module will disclose its general behavior (do large I/O, for example), and it is left to the discretion of the system how to best provide service for that behavior. The system might choose a different scheduling algorithm for that task, choose to implement its I/O using an implementation weighted towards VM techniques, or ignore it entirely.

Since these annotations describing properties of a module are advisory, it is up to the system to decide how best to use this information to provide better services. Since it is not built into any client-visible interfaces, the system has the ability to completely change its approach without the burden of maintaining backwards compatibility. Furthermore, the client of the service is not burdened with the portability limitations inherent in importing extended interfaces to influence system behavior. In this system, implementation is not destiny.

In Section 2 we discuss the difficulties constructing simple, expressive interfaces. In Sections 3, 4, and 5 we discuss the OMOS server, its approach to specialization, and active *electric libraries*. In Sections 6 and 7 we demonstrate several uses of OMOS and a sample I/O library to optimize program execution.

2 Problems with System Interfaces

2.1 The Evolution of Operating System Interfaces

Some operating system interfaces exemplify the problems of poor modularity and poor encapsulation. Early IBM operating systems such as EDX on the Series-1 provided very little encapsulation; later IBM operating systems (OS/360, MVS) evolved to support many different interfaces, highly specific to the different services they provided. Different file access methods (e.g., ISAM, VSAM, myriad tape routines, etc.) were required to process different forms of data. These interfaces required the application programmer to specify a great deal about the data to be used by their program, rendering it difficult to take advantage of underlying commonalities in different data types.

Focus changed as operating systems evolved. The Unix[23] operating system was an example of a significant departure in the nature of interfaces, due, in part, to its use of a simple, universal I/O interface. Under Unix, files and devices are both assigned entries in a common namespace and accessed via a small set of operations. In Unix programs there tends to be little information encoded in a program relating to I/O that would impede its portability. For the most common operations (e.g., `read`, `write`, `seek`), Unix provides a uniform mechanism over a wide variety of data types. Programs that use these simple I/O primitives can be applied to many different types of data, and tend to be portable to many different variants of Unix.

Unfortunately, simple interfaces provide little information for the system to use in order to guide its operation. If, for example, a program is to plan to read entire files sequentially, this knowledge is very helpful for an operating system planning its caching strategy. With little or no information about the nature of each client they service, operating systems must often restrict themselves to algorithms that provide reasonable performance for the majority. For example, the LRU caching algorithm works quite well in the general case and disastrously in the case where a large amount of data (greater than the cache capacity) is scanned sequentially.

The intended use of an interface by a program can sometimes be intuited or derived by the operating system, but this process can be expensive or error-prone. Others have discovered, for example, that one of the strengths of the Unix `read` call is that its code path is very short in the case where small amounts of data are being read from the Unix buffer cache. Inserting code in this path (such as would be required for monitoring or metering to intuit application intent) risks significantly affecting the performance of programs that work within the common-case parameters of `read`[24].

2.2 Example: Memory Allocation Interfaces

Storage allocators are also good examples of facilities that must strike a delicate balance. In most implementations of the Unix library storage allocator (`malloc`), the operation to allocate storage (`malloc`), takes just a few instructions when an appropriately-sized block is available. The operation to release storage (`free`), takes just a few instructions. The only work done is to derive the freed block's size and thread it onto a linked list; blocks are never coalesced and memory is never returned to the system.

Weighting allocations towards commonly used block sizes, or towards coalescing, or towards deallocating free blocks, are all examples of optimizations important to some applications. We have found, as have others[10], that adding just a little code to choose effective strategies for `malloc` clients that don't fit the common case, expands the simple code path and significantly hurts the performance of all other applications. An extreme case of this sort of "program slowdown through optimization" comes with short-running programs that allocate a lot of space, and then exit. For these programs, any effort to anticipate clever reuse of storage, or, in general, any investment in speeding up the program later in its run, is wasted.

The most frustrating aspect of this tradeoff is that the application developer typically knows in advance how the program is going to use its resources. However, there is usually no means to communicate this information to the system.

An ad hoc compromise is sometimes struck whereby interfaces are extended to include operations that, effectively, specialize the implementation of that interface at run time. The HP-UX implementation of the Unix routines `mallopt` and, to some extent, `madvise` in several Unix systems, are examples of this sort of compromise. `Mallopt` can be used to specify the application's preferred bucket sizes, whether malloc should lock out signal handlers while manipulating sensitive data structures, rounding criteria, etc. `Madvise` is used to control the paging style to be used (`madvise` is not entirely applicable since it works on a per-region basis).

The approach of providing extensions to interfaces is not satisfactory, however, since it pollutes the application program with information about library implementations. Moreover, applications that take advantage of `mallopt` must be conditionally compiled on systems where it is not available. `madvise` is not implemented on many systems, yet it is part of most library interfaces. . . just in case. In addition, systems that use extended interfaces to specialize service implementations are forced to respond to these service requests at run time[2], when the information could be used at compile or link time to generate a better application. As it stands, unless the implementation uses dynamic linking, all possible code paths must be present within the application. Specialization can reduce the number of code paths present in an application in many cases.

A better approach to the problem of bridging the gap between clients and their service providers is to give clients the means to describe relevant attributes of their program, without exposing details of system implementation or complicating interfaces. One crude means of partially attaining this goal is the use of multiple libraries. Users specify, at the level of the application build environment, what functionality they want, by choosing implementations from different libraries. HP-UX does this with `malloc`: they offer an alternate `bsdmalloc` library which attains speed at the expense of space[11]. On the SunOS and VR4 operating systems, some crude run time control is available by modifying the LDPATH and LD_PRELOAD environment variables to modify the order in which libraries are searched [9]. The problems with these approaches are two fold: (i) they offer only a coarse granularity of control, and (ii), the system has been forced to export some knowledge of its implementation to application builders, setting their expectations and constraining system development. While applications would not have the knowledge of system implementation built into them, proper, it would be littered throughout their build and execution environments.

3 The OMOS Approach to Specialization

OMOS provides another way to solve these problems, putting more of the burden and control in the hands of those providing system services. The approach taken within the OMOS system is threefold:

- OMOS allows modules to be active entities (referred to as *electric libraries*) that abstract over their implementation, capable of making decisions on their own. Rather than using dead, lifeless archive files for libraries, OMOS libraries are implemented as functions. The client module needing a library is given to the OMOS *electric library* as an argument. The service provider (as represented by the library) can then provide different implementations, based on computations it performs.

- OMOS provides a means of annotating modules, so that application developers have a way to disclose *what* a client does— its high-level semantic attributes— rather than just *how* it does it (how it operates is determined by the interfaces it imports).

- OMOS provides a run time connection to the underlying Unix operating system, so that static information (inferred and disclosed) about the application can be used for additional specialization at run time.

Specialization involves developing an implementation tailored to a given set of operating conditions; the application, libraries, and even operating system services can be specialized using the OMOS approach.

In all cases, OMOS can specialize services that would otherwise either be specialized at a later binding time (e.g., run time) or that could not be specialized at all, forcing the application to use a more general, less efficient implementation.

[2] In the case of `mallopt`, calls to it must be made before the first "small block" is allocated.

When specializing programs, OMOS plays either a direct or supporting role, depending on the type of transformation indicated. OMOS directly implements link-time specializations, such as the `malloc` specialization described below in Section 6.1. For other types of specialization, OMOS generates a description of the program's properties and introduces code to transmit, at program startup time, that description to the Unix server. The Unix server makes use of the information, together with its knowledge of the actual run time environment, to guide its own run time decisions on providing services to the program.

As mentioned above, OMOS clients structure annotations such that they remain *advisory* only. Which set of annotations to recognize and what effect this recognition will have is entirely at the discretion of the service implementor. It is assumed that annotations will signal opportunities to *specialize* a generic service, narrowing its focus or redefining the expected common case. But the service is always free to ignore the annotations and produce a generic implementation.

OMOS does not define the particular attributes that are covered by module annotations; instead, OMOS provides enabling technology. In order to make effective use of this technology, it is still necessary that clients and service providers agree on a common vocabulary for describing and responding to important modes of operation (e.g., "runs a long time," "does I/O in large chunks," "does sequential I/O," "does random I/O," "is a filter," etc.). Individual decisions a service provider may make, in responding to a client's annotations, may have the implementation-specific flavor found in extended interfaces such as `mallopt`. But, this detail is maintained internal to the service provider and does not soil the pristine client. Should `mallopt`, or the equivalent, cease to be the correct approach to optimize a service, the provider is free to change its approach as needed.

4 The Design of OMOS

The work described in this paper used OMOS running on a Mach 3.0[17] system which provides Unix emulation through a Unix server process. OMOS is used by the Unix server to load programs and shared libraries. OMOS has also run on standard Unix systems, such as HP-UX, where we studied more conventional aspects of OMOS's shared-library service[18].

OMOS provides a level of indirection between a module name and its implementation. In most operating systems, programs or dynamically loaded modules are implemented using files. As a result, the system has little or no control over their implementation: their nature is fixed at install time. In contrast, OMOS exports a fine-grained module namespace. Clients, such as the Unix `exec` routine, that obtain program code from OMOS, request that code by *name*. OMOS determines the specifics of what code the client receives.

The OMOS namespace includes elements such as code fragments (e.g., Unix "dot-o" format files), and *module specifications* (historically referred to as *meta-objects*). Module specifications are small extension language programs whose execution results in the generation of code fragments. As a result, all elements in OMOS' name space are modules of one form or another.

In order to translate a module name into an implementation, a client (e.g., the Unix server) asks OMOS to resolve the module name. OMOS evaluates the associated module specification to produce an implementation which is then cached (to disk). Further requests for the same module name will be satisfied from OMOS' cache.

If the name is the name of a code fragment, or, as is usually the case, it names a module specification for which a cached copy of its implementation exists, OMOS returns a pointer to that fragment immediately. If not, OMOS evaluates the module specification, producing an implementation; typically, it caches the result (on disk) and returns a pointer to that implementation.

OMOS uses the STk[8] implementation of Scheme[6] as an extension/scripting language. Low level

```
        PSEUDOCODE                            |          VERBATIM SCHEME CODE
                                              |
/* crt0 & libc happen to be FUNCTIONS */      |  ;; lookup and load library implementations
startup = load("/lib/crt0")                   |  (let ((crt0 (resolve "/lib/crt0"))
clibrary = load("/lib/libc")                  |        (libc (resolve "/lib/libc")))
/*                                            |
  Apply the functions (libraries) to our      |
  module.  I.e., first resolve ls.o's         |
  external references from the C library,     |
  and then resolve any more external          |
  references from the startup library.        |
*/                                            |  ;; apply them to our module
startup(clibrary(load("/obj/ls.o")))          |  (crt0 (libc (resolve "/obj/ls.o"))))
```

Figure 1: Sample Module Specification: */bin/ls*

object-file manipulation and other primitive OMOS operations are implemented in C and C++ We have extended STk with datatypes that represent OMOS module specifications and code fragments, and with a set of operations designed to manipulate object file symbols and interfaces. These operations are based on a formal model described in the Jigsaw language[1, 2], that provides a basis for relating modularity and inheritance. Higher-level (and more sophisticated) module manipulation, such as is done in a *module specification*, are implemented with Scheme scripts.

With its extension language, OMOS can perform arbitrary transformations on modules. The system can use these module operations to *wrap* procedures (in order to augment procedure functionality), to *replace* procedures, to *unbind* or to *overload* procedure names. OMOS also provides operations that allow users to generate modules on the fly from source, and manipulate groups of defined or undefined symbols, specified using regular expressions. Using these operations, the standard Scheme facilities, and the persistence of its store, OMOS can perform conditional linking and adaptive transformations on modules.

Figure 1 demonstrates a simple module specification that links a Unix application (`ls`) with implementations of the system *electric libraries* `crt0` (the standard "C Run Time" startup code) and `libc` (the standard C library). In this example, `crt0` and `libc` are both functions (lambda expressions); `/obj/ls.o` is an code fragment. The `libc` function is invoked on `/obj/ls.o`, causing the unresolved symbols in `obj/ls.o` to be searched for in the `libc` code fragment, and, to whatever extent possible, resolved. The result of that operation (a module) is passed to `crt0`, which completes the link.

5 OMOS Electric Libraries and Module Annotation

As was mentioned above, OMOS libraries are often implemented as functions: Scheme lambda expressions known as *electric libraries*. One of the principal advantages of this technique is that libraries may shift their content, based on the module against which they are being linked.

Electric libraries can infer substantial semantic information simply from symbolic names. For example, if neither of the symbols "fork" or "exec" is referenced by the target program, then the program will not spawn a child process. The library can then use that information to select specialized implementations of its routines. (Section 6.2 discusses specializations based on the usage of "fork/exec".)

There are other kinds of module attributes that are not evident based only on the module's interface. Therefore, OMOS supports module *properties*. Properties are simply a list of attributes that can be declared

```
(let ((crt0 (resolve "/lib/crt0"))
      (libc (resolve "/lib/libcsmart")) ; a smart version

      (cat  (properties '(big-io) ;; THIS IS THE ANNOTATION
                        (resolve "/obj/cat.o")))))
  (crt0 (libc cat)))  ;;actually link them
```

Figure 2: Annotated Unix "cat" Program: */bin/cat*

```
(lambda (m)   ;this whole script returns a function taking one arg (m)
  ;; name our modules and libs
  (let* ((common-list '("/obj/printf.o" "/obj/scanf.o" "/obj/stat.o" ...))
         (kern-fds    '("/obj/open.o" "/obj/read.o" "/obj/write.o" ...))
         (user-fds    '("/obj/u_open.o" "/obj/u_read.o" "/obj/u_write.o" ...))
         (spawn-syms  '("fork" "exec" "system"))
         (no-spawn    (or (has-property? 'no-spawn m)
                          (not (sym-referenced spawn-syms m)))))

    (merge m          ;construct the libc function
      (ar-project m   ;'ar' as in Unix .a "archive" file
        ;; merge "m" with the modules it references, special casing a few
        (map resolve
          (append common-list    ;start appending to the common .o modules
            (if no-spawn  ;if can append a special user i/o lib
              user-fds    ;then user i/o
              kern-fds)   ;else kernel i/o
            (if (has-property? big-io m) ;if does big-io
              '("/obj/page_malloc.o")    ;then page-aligned malloc
              '("/obj/malloc.o"))))))))  ;else normal malloc
```

Figure 3: Simple Libc Transformation: */lib/smartlibc*

when the module is defined, and tested wherever a module is referenced. Properties are typically semantic attributes disclosing information about the expected behavior of the module. Figure 2 shows a version of the Unix `cat` program that has been annotated to include one property: that it does I/O in large chunks.

In a more complex example, figure 3 shows a version of the Unix C library that has been programmed to respond to the properties `no-spawn`, and `big-io`. The standard operation of the C library is to extract the closure of modules that *m* references; the module *m* is then merged with that list, resolving those symbols. In this example, the functionality is extended so that if *m* has either of the properties `no-spawn` (does not fork: either inferred or disclosed) or `big-io`, this results in being merged with different versions of the I/O and storage allocation routines.

Properties are currently defined to be inherited across all operations; merging two modules results in a new module that reflects the union of the properties found in the two component modules. We envision the need for a richer set of rules for combining and inheriting properties, resolving conflicting property values, and associating more semantic attributes with properties. Eventually we will develop those, but for the moment, this simple definition provides reasonable functionality.

6 Program Specialization Using OMOS

In the next sub-sections we describe several specific specializations within the context of the Mach-based OSF/1 Unix system. We report on the performance results of some of these specializations later, in section 7.1.

6.1 Configuration Management

Often, a given interface can face a range of operating conditions, each of which requires a different implementation to function both correctly and efficiently. If a library developer can't accurately predict or detect those operating conditions, the developer needs to produce an implementation that works in all cases. This is frequently more complex, and almost always less efficient. On the other hand, if a range of implementations *do* exist, having the application developer specify the appropriate version "by hand" is an error-prone process, due to semantic subtleties and, especially, program evolution. By contrast, libraries developed within OMOS can use properties and symbol values to simplify this process.

For example, in multi-threaded systems, programs that use multiple threads require special versions of a number of standard routines (e.g., `malloc`) in order to avoid corruption due to locking omissions. Under typical Mach implementations, the Unix `fork` routine generates a single-threaded child. As a result, `fork` is implemented using a special pre-fork and post-fork that seize and release all malloc locks (respectively) in order to assure that the child process's storage allocation state is known on startup.

Programs that do not make use of multiple threads, including the vast majority of Unix programs, should not have to concern themselves with such details. Figure 4 demonstrates a version of an OMOS library exporting `malloc` that automatically produces the correct version of `malloc` and `fork` depending on whether the client being linked into it is multi-threaded or not.

In addition, if the program is declared to have the property `long-running`, this malloc library will automatically link in routines that will periodically compact and deallocate unused memory, reducing the virtual memory size of the program. This feature is unneeded and wasteful in programs that run for a short time and exit, but is very valuable in long-running programs, such as servers, that allocate and free substantial quantities of memory. Using a compacting allocator reduces the consumption of swap and address space and increases locality.

We have implemented these versions of malloc, have found find substantial performance differences between them. For example, our malloc/free package for single-threaded programs achieves a factor of nine speedup over the vendor-provided HP-UX and SunOS malloc routines, while our multi-threaded version is successful in reducing VM consumption, but is not as fast as the single-threaded version. [Note to reviewers: we are currently choosing long-running programs in which to evaluate this tradeoff. For the final paper we will have performance results for OMOS-specialized servers and non-servers. In particular, we will try specializing OMOS itself, and report not only on the results of specialization, but on OMOS's own memory allocation behavior.]

6.2 I/O Specialization

The Unix I/O interface falls into the category of an interface that often forces its implementors to produce sub-optimal instances. As we shall see, it is desirable to be able to implement portions of the Unix I/O system in user-space libraries. Some optimizations are facilitated if I/O is implemented in user space, such as the use of VM primitives to supplant I/O, reducing the amount of time applications spend interacting with the operating system.

The heart of the Unix I/O mechanism is the file descriptor table. It serves as a switch that converts a

```
(lambda (m)   ;this whole script returns a function taking one arg (m)
  (if (sym-referenced "cthread_init" m)   ;if multi-threaded

    ;; then select a multi-threaded malloc, and check more things....
    (merge
      (resolve "/obj/mthr-malloc.o")

      ;; if we are long-running,
      (if (has-property? 'long-running m)
        (resolve "/obj/mthr-compact.o")  ;add compaction module
        '()) ;else do nothing

      ;; if we reference fork,
      (if (sym-referenced m "fork")
        ;; then redirect calls to fork to its multi-threaded version
        (merge (rename '("fork") '("m_fork") m)
          (resolve "/obj/mthr-fork.o"))
        m)      ; else return module we've built: 'm'


    ;; else not multi-threaded, so select non-threaded malloc
    (merge m (resolve "/obj/malloc.o")))
```

Figure 4: Program Configuration: */lib/malloc*

small integer handle into an object descriptor on which various I/O methods are defined. The semantics of the Unix process spawn operations, `fork` and `exec`, significantly complicate implementation of user-space versions of these operations, however.

The `fork` operation, which creates a duplicate child address space is defined as establishing a shared copy of the parent's I/O state in the child. Operations on one process's file descriptors are expected to change the state (in particular, the file offset location) of the other process's file descriptors. In order to maintain full Unix file descriptor semantics after a `fork`, either the file descriptor table needs to be maintained by a common third party (such as the Unix kernel or server process), or there needs to be a complicated state transferral and shared state update mechanism.

The Unix load primitive `exec` is defined as wiping out the child process's memory and replacing it with a new program image. The child's I/O state is expected to remain across the `exec`, however. If that I/O state is implemented as a naive user-space library it will be wiped out along with the rest of the previous occupant's state. Certain portions of memory may be marked to be preserved across `exec`, which alleviates the direct impact of this problem. However, *all* components of the I/O system must only use memory resources that are maintained across exec; any I/O manager that uses the system `malloc` routine to allocate memory risks having that memory wiped out at the next `exec`; any I/O manager that uses resources that, in turn use `malloc`, such as the standard I/O library, are at risk; and so on.

The vast majority of Unix programs do not spawn other processes, however, so it is pessimal to restrict the implementation of an I/O subsystem to the needs of those that do spawn. In keeping with the OMOS philosophy of picking specialization opportunities prudently, OMOS permits using a specialized user-space I/O subsystem (and, in turn, specialized I/O streams), only with those programs that cannot, even potentially, make use of the `fork` or `exec` system services. As was mentioned earlier, Figure 3 is an example of a version of the Unix C library that has been programmed to provide different implementations of the standard file descriptor operations, depending on whether a user-space implementation is possible or not.

Given a user-space implementation of I/O operations, many optimizations of existing I/O operations are enabled or facilitated. In the server-based versions of Unix implemented on top of Mach, there is a penalty that is paid for accesses to the Unix server; it has been demonstrated that, on average, system calls in a server-based world require more processing than equivalent system calls in traditional in-kernel implementations[5]. Passing information between protection domains in messages involves additional work than when using straightforward trap-based system calls. So, to improve performance, we seek out mechanisms that, among other things, permit us to circumvent the Unix server. A number of variants are explored in the next section.

6.3 Specialized User-space I/O Managers

Part of our user-space I/O implementation is a user-space file descriptor table; each valid file descriptor points to an instance of a given I/O class. The system supports several different I/O classes, each of which is derived from an abstract base class that defines Unix I/O operations, `read`, `write`, `seek`, etc.

The various derived classes implement strategies appropriate to different performance tradeoffs. The simplest I/O class passes its operations through to the Unix server; this class provides the default behavior for cases where there is no straightforward optimization. More sophisticated I/O classes perform I/O using VM operations or other strategies that perform well under specific, commonly occurring conditions.

The I/O classes used by a program are chosen based on the nature of the data being accessed and properties of the application accessing them. For example, our results detailed in Section 7.1 show that for programs that tend to do small sequential reads from small files, it is advantageous to map the entire file into memory and perform memory-to-memory copies from the mapped area into the user's buffer. By contrast, for small reads from *tiny* files, it may be advantageous to do I/O via a buffer that is shared with the Unix server, so that the overhead of mapping the file and doing page-at-a-time I/O do not overshadow the benefits gained from avoiding Unix system calls and working out of the memory system's page cache. For large sequential reads from large files, it is advantageous to use VM operations to map pages of file data directly into the user's read buffer; particularly in the case where the user's buffer is page-aligned and the reads are done in even page multiples. In cases where I/O is done in large chunks, it helps to further specialize the application to use a special storage allocator that returns page-aligned buffers.

Neither the core operating system nor OMOS can know, *a priori*, what kinds of I/O an application will tend to do, even though that behavior may well be identical across all invocations of a given application. For example, the standard version of the OSF/1 `cat` program (which reads and concatenates the contents of a number of data sources) has a fixed buffer size. The read requests it issues are always for 8192 bytes. While this information may vary from application to application, or even version to version, if there are cases where it is consistently applicable, this is information that can be used to produce a better read path.

However, even if information such as this is known ahead of time, it is possible the actual run time environment may not allow such specialization. For example, the I/O may be directly from a device such as a serial line, where VM mapping will not work. When such a possibility exists, the ultimate specialization decision must be deferred to run time. The program still needs to disclose its expected behavior, however. In order to communicate this behavioral information between the application and the operating system, OMOS encodes into a string some of the application's operational properties, both inferred and disclosed (e.g., does-big-io, no-spawn, . . .). OMOS also arranges for the application's startup code automatically to register that string with the Unix server (at exec-time). In this way, the application can guide the Unix server's treatment of its subsequent open requests.

For example, OMOS may arrange for a program to register itself as an application that does sequential (or random) I/O, or as an application that typically does I/O in small (or large) chunks. When the Unix server receives a request to open a file or device, it uses this application-provided information to guide its choice of

I/O object to return. Figure 5 shows the algorithm our modified Unix server uses to choose the appropriate type of I/O object to return for file read operations[3]. Section 7 will discuss how we arrived at that algorithm.

The decision as to which I/O object type to return is centralized in the Unix server, rather than built into the user-space library. In this way, the decision is made at the location where the most information is available (e.g., file/device attributes, past process behavior, etc.)

7 Results and Status

7.1 Results

All performance results were gathered on a 67Mhz HP 730 (PA-RISC architecture), with 64 megabytes of main memory and a 256K split I/D cache, running Mach 3.0 and an OSF/1 1.0 server. Each test was repeated three times, and exhibited negligible variance.

I/O Managers: We evaluated some of the tradeoffs involved in different I/O types, under different patterns of use. The results are summarized in Tables 1 and 2. The results demonstrate that, depending on conditions, different I/O strategies are optimal. A single strategy was not always superior, offering the opportunity to profit from intelligent specialization. The I/O strategies we implemented included:

1. *Kernel I/O:* This strategy, the default, passes all operations through to the Unix server, performing no local processing.

2. *Small I/O:* This strategy is optimized to work best with small I/O transfers, which re-read, or randomly read, the data within a given block. It takes advantage of a block of memory shared between the application and the Unix server. The Unix server services small I/O requests by copying its buffer cache into the shared area, reducing the cost of transferring the data by avoiding transmitting the data through the kernel in a message.

3. *Page I/O:* This strategy uses VM operations to reduce message traffic to the kernel. For each read request, Page I/O maps the appropriate region into the user's address space. If the user's buffer is page-aligned, the Unix server performs a virtual copy operation, using the data pages underlying the mapped region to back the memory in the user's buffer. In the best case, this operation permits data transfer without actual copying. In the case where the user's buffer is not page aligned or if some portion of the copy is less than a page in length, the data are copied directly from the mapped region into the user's buffer (which, while not as efficient as the ideal VM copy, still avoids user$\leftrightarrow$kernel message traffic. Since this strategy does not map the whole file, for very large files it avoids what would otherwise be excessive use of address space.

4. *Whole File I/O:* This strategy maps the entire file into memory and physically copies the requested portion from the mapped region to the user's buffer. This strategy avoids the overhead of remapping the buffer and works well when there are large numbers of random or small I/O operations.

The data in Tables 1 and 2 were produced by running a program that does variable sized input on files of specific sizes.

[3] File write operations involve a set of interesting tradeoffs on their own. We have not yet begun to experiment with I/O objects to specialize file write operations.

Table 1: Simple Read: Seconds for 10000 Iterations of open/read/close

	1 byte file			8k byte file			32k byte file			128k byte file		
read size	1k	8k	wf	1k	8k	wf	1k	8k	wf	1k	8k	wf
Kernel I/O	11	**10**	**10**	32	**14**	**14**	103	34	58	432	154	224
Small I/O	**10**	**10**	**10**	31	**14**	**14**	100	**33**	58	418	153	223
Page I/O	11	11	11	32	16	15	103	36	**10**	391	115	**34**
Whole-map I/O	11	11	11	**24**	15	15	**68**	34	35	**246**	**106**	91

Table 2: Reread: Seconds for open/10000 reads/close

	1 byte file			1k byte file			8k byte file			32k byte file		
read size	1k	8k	wf	1k	8k	wf	1k	8k	wf	1k	8k	wf
Kernel I/O	**3**	3	3	**3**	4	4	**4**	24	7	**7**	96	**26**
Small I/O	**3**	3	3	4	3	3	7	23	7	52	**92**	**26**
Page I/O	**3**	3	3	**3**	4	4	7	24	7			
Whole-map I/O	**3**	**1**	**1**	4	**1**	**1**	7	**12**	**6**			

Each entry in table 1 represents the time in sections to complete 10,000 iterations of an open/read-to-completion/close loop. Additionally tests with a 1k byte file were also run but always produced the same result (11 seconds) regardless of the read size and I/O strategy. Each entry in table 2 represents the time in seconds to complete a single open, followed by 10,000 read/re-read iterations and a single close. The tests were run with a hot cache; kernel I/O uses the Unix buffer cache directly, while VM operations work from the Mach memory object cache.

The highlighted entries represent the I/O strategy(s) that performed best for a specific file size and read size. The data clearly shows the size of the read and the size of the input file has a dramatic effect on the utility of a given strategy. Based on this data, we programmed the Unix server as shown in Figure 5, so that it returns, on `open`, an appropriately specialized I/O object.

If the module indicated in an annotation that it did "big I/O," this indication is transmitted to the Unix server at startup time. The information is used in subsequent open requests to guide the decision of the Unix server when choosing the user's I/O strategy. Selection of the VM-based I/O methods also results in the application being linked with a version of `malloc` that, when possible, returns page-aligned buffers.

The parameters we selected in this I/O strategy are approximate. They primarily serve to demonstrate an example of the fine-tuning that is available to the system implementor who has OMOS at his or her disposal.

Automatic specialization: To demonstrate that OMOS and the Unix server can successfully choose specializations, and that they can succeed in improving performance, we tested this approach on a few small Unix programs. `grep`, which searches for string patterns in files, was run over 57 files ranging in size from 13K to 3MB, totaling 42MB of data searching for the string "foobar" in each file. When `grep` was annotated and run under OMOS (Page-IO was chosen by the Unix server), its elapsed time was reduced by 6%. Using the same input files, a similar test was run on the `wc` program, which counts bytes, words and lines in files. The elapsed time for `wc` improved by 27%, in large part due to reduced copying by the operating system. Our final test was to run `cat` to concatenate a numerous input files (618 files with approximately

```
medium_big = 32 * 1024;         /*  32K is max file size using small i/o */
max_whole_map = 128 * 1024;     /* 128K is max file size that maps whole */

if (file_size < page_size)
    return opentype_small_io;
else if (file_size <= medium_big) {
    if (user_does_big_io)
        return opentype_whole_map;
    else
        return opentype_small_io;
} else if (file_size <= max_whole_map) {
    if (user_does_big_io)
        return opentype_page_io;
    else
        return opentype_whole_map;
} else                          /* very big file */
    return opentype_page_io;
```

Figure 5: Open-time I/O Strategy Selection

Table 3: Utility Timing Breakdowns (average time in seconds over three runs, hot cache)

	application user	application system	server user	server system	wall clock
Standard grep	2.9	0.1	2.1	0.2	5.4
Specialized grep	3.3	1.6	0.1	0.1	5.1
Standard wc	9.2	0.9	1.9	2.5	14.6
Specialized wc	9.0	1.5	0.1	0.1	10.6
Standard cat	0.8	0.1	3.1	1.6	20.6
Specialized cat	0.6	0.8	1.2	0.7	11.0

17MB of data). After annotation `cat`'s elapsed time improved by 47%. More detailed breakdowns of the test results can be found in Table 3.

The very large improvements `cat` seems to indicate that programs which perform minimal processing on large amounts of input data benefit the most from the current I/O specializations. [The final paper will include study of more programs of this nature.]

This approach has the weakness that it is applied to *all* I/O a program does through this mechanism. Although I/O can be customized on a module-by-module basis, potentially providing each module with a different implementation of the basic I/O primitives, interaction between modules usually precludes this in a practical sense (e.g., if file descriptors are implemented to mean different things in different modules, passing file descriptors between modules via other (uncontrolled) parts of a module's interface becomes problematic). Despite this weakness, this I/O implementation does demonstrate the potential of producing significant speedups and provides a useful starting point for experimenting with user-space I/O optimization strategies.

7.2 OMOS Status

As indicated, OMOS is a server written in C and C++, incorporating STk as its extension/scripting language. OMOS runs on Mach 3.0, with both the OSF/1 and CMU server-based Unix implementations, and on both the HP PA-RISC and Intel x86 platforms. To manipulate object files, OMOS uses the GNU BFD[4] object file library to provide it with a high degree of independence from low level object format details. OMOS's abstract persistent store, used to cache linked modules, is currently implemented using the Unix file system. In addition to the server reported here, we also have a non-server variant of OMOS that provides its linking, module, and object file symbol management functions as a normal Unix application. We have found this highly useful for a variety of everyday uses. OMOS currently consists of approximately 17,000 lines of code, excluding STk (11,000 lines) and the BFD library.[4]

In prior work in an earlier version of OMOS that lacked a full scripting language, we demonstrated that OMOS's shared library services are fast and flexible[18]. Other work demonstrated that OMOS could provide transparent call-graph profiling and re-ordering of executables at the granularity of functions, providing speedup due to improved code locality[20]. OMOS could do this quite easily, due to its mediating role in program invocation, its active nature, its detailed knowledge of object files, and its powerful module manipulation primitives.

8 Related Work

The approach of using "open implementations"[15] to avoid the performance and flexibility limitations of so-called "black-box abstractions" has recently received a lot of attention. Meta-object Protocols[14] take an imperative, directive approach to customizing implementations. That work was first applied in the compiler domain, where the compiler implementation was customizable by users. The approach has been applied in the operating system domain as well; user customization of virtual memory management is an area that has historically received a lot of attention. Some of the most recent VM work, which takes a more incremental approach than, for example, external pagers, is that by Anderson et al[12]: the OS makes upcalls to user code which directs the OS how to manage individual pages on its behalf.

By contrast, in the I/O domain, the work on Transparent Informed Prefetching[21] uses *disclosure* of information about an application's pattern of I/O use to allow the underlying system to pre-fetch file data.

The majority of the OS open implementation research has been focused on the operating system proper itself. We believe our approach of specializing the user-space library, extracting semantic information from application's symbolic names, and adding external annotations to the application when needed, is unique.

Maeda's work[16] on user-space implementation of network protocols is related to our I/O library examples, in that responsibility is shared with the kernel. The Apollo type manager implementation[22] is an example of a file descriptor-based system for doing Unix I/O, implemented largely in user-space, communicating to the OS only when security requirements dictated.

9 Future Work

We plan to apply our work to other domains besides reducing copying by user space I/O libraries. In particular, in Cao's work[3] on informing the OS so it can use more appropriate buffer management strategies, we have an area where our transparent approach will likely apply and show significant performance benefits.

[4] The entire source and binaries for a stable version of OMOS, running on the Mach kernel and Lites Unix server, on both Intel x86 and PA-RISC platforms, will be packaged and available for anonymous ftp by December 1st, 1995.

In prior work we demonstrated OMOS doing static call graph analysis. This can be used to determine the call chain of particular service invocations, and allowing us to avoid the weakness of this current work, that all I/O through a particular interface is treated identically.

10 Conclusion

The OMOS linker/loader system provides an intelligent mechanism for communicating information from an application to its imported libraries and to the operating system. This information is crucial for driving application, library and operating system service specializations to improve application runtime performance. We have improved some small UNIX programs by 6% to 47% in a Mach/OSF-1 environment without polluting applications.

While the oddities and weaknesses we have chosen to attack are specific to this application and/or environment, they are analogous to attributes and tradeoffs found in other applications and/or environments. These oddities and weaknesses are also representative of the gaps between what we say, what we are allowed to say, and what we mean, when constructing a system. By creating a mechanism that allows passing detailed description of intent between a client application and the services it uses, we provide the means to build better, more efficient systems without losing the benefits of abstraction and modularity.

Elevating the system linker/loader to be a first class member of, and fully integrated with, the operating system, gives a powerful tool. Together with the level of indirection provided by having all executable programs be scripts, we have a system which is more flexible and extensible than existing systems.

Acknowledgements

We are grateful to Guru Banavar for his contribution to the Scheme support, our understanding of module manipulation, and his help in reviewing this document.

References

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

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

[3] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 165–177, Monterey, CA, Nov. 1994. USENIX Assoc.

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

[5] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133, 1993.

[6] W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), 1991.

[7] P. Druschel. Efficient support for incremental customization of OS services. In *Proc. Third International Workshop on Object Orientation in Operating Systems*, pages 186–190, Asheville, NC, Dec. 1993.

[8] E. Gallesio. Embedding a Scheme interpreter in the Tk toolkit. In L. A. Rowe, editor, *First Tcl/Tk Workshop, Berkeley*, pages 103–109, June 1993. Included in ftp://kaolin.unice.fr/pub/STk-2.1.6.tar.gz.

[9] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in SunOS. In *Proc. of the Summer 1987 USENIX Conference*, pages 131–145, Phoenix, AZ, June 1987.

[10] D. Grunwald and B. Zorn. Customalloc: Efficient synthesized memory allocators. *Software — Practice and Experience*, pages 851–869, Aug. 1993.

[11] Hewlett-Packard. *HP-UX Release 8.x Manual Set*, 1990.

[12] A. V. K. Krueger, D. Loftesness and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Oct. 1993.

[13] G. Kiczales. Foil for the workshop on open implementation. http://www.xerox.com/PARC/spl/eca/-oi/workshop-94/default.html, 1994.

[14] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[15] G. Kiczales and J. Lamping. Operating systems: Why object-oriented? In *Proc. Third International Workshop on Object Orientation in Operating Systems*, pages 25–30, Asheville, NC, Dec. 1993.

[16] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, 1993.

[17] Open Systems Foundation and Carnegie Mellon Univ. *OSF MACH Kernel Principles*, 1993.

[18] D. B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and flexible shared libraries. In *Proc. of the Summer 1993 USENIX Conference*, pages 237–251, June 1993.

[19] D. B. Orr and R. 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.

[20] D. B. Orr, R. W. Mecklenburg, P. J. Hoogenboom, and J. Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proc. of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993.

[21] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proc. Third Intl. Conference on Parallel and Distributed Information Systems*, pages 28–30, Austin, TX, Sept. 1994.

[22] J. Rees, P. Levine, N. Mishkin, and P. Leach. An extensible I/O system. In *Proc. of the Summer 1986 USENIX Conference*, pages 114–125, June 1986.

[23] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1930, July/August 1978.

[24] M. T. Stolarchuk. Faster AFS. In *Proc. of the Winter 1993 USENIX Conference*, pages 67–75, San Diego, CA, 1993.