1

DYNAMIC PROGRAM MONITORING AND TRANSFORMATION USING THE OMOS OBJECT SERVER

Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom and Jay Lepreau

Department of Computer Science
University of Utah, Salt Lake City, Utah 84112
USA

ABSTRACT

In traditional monolithic operating systems the constraints of working within the kernel have limited the sophistication of the schemes used to manage executable program images. By implementing an executable image loader as a persistent user-space program, we can extend system program loading capabilities. In this paper we present OMOS, an Object/Meta-Object Server which provides program loading facilities as a special case of generic object instantiation. We discuss the architecture of OMOS, the extensible nature of that architecture, and its transparent application to the problem of dynamic program monitoring and optimization, in particular, code reordering to improve locality of reference. We describe five reordering strategies, one using the static call graph and four based on analysis of the dynamic call graph. One large application was studied under varied amounts of available memory. The best dynamic strategy gave speed improvements ranging from 23 to 66%, while the static method gave speedups ranging from 19 to 52%.[1]

[1] This research was sponsored by Hewlett-Packard's Research Grants Program and by the Defense 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 the Defense Advanced Research Projects Agency, the U.S. Government, or Hewlett-Packard.

1 INTRODUCTION

Traditional program linking and loading facilities, like those found in Unix[18], have simple semantics. They were developed in a time of small address spaces and small applications, often with the goal of minimizing linking time and use of kernel resources. These linkers and loaders tend to use simple external structures — executable files, libraries, etc. — to reduce kernel complexity. One consequence of the simplicity of implementation is that, as architectural advances free programs to grow in size and complexity, the linking and loading algorithms do not scale well. The resulting programs produced often exhibit poor locality of reference characteristics. In addition, program loading and execution facilities tend to be separate from compilation facilities, making it inconvenient to perform optimizations based on information derived at run-time.

In this paper we investigate the use of OMOS, an Object/Meta-Object Server, to improve locality of instruction reference by dynamically monitoring and transforming executable images. We begin by discussing typical linker technology and the particular problems of maintaining locality of reference within large programs. We next provide an overview of OMOS, its general organization, and its object loading facilities. Subsequently, we describe the use of OMOS' extensible nature to transparently monitor and transform executables to improve locality of reference. Finally, we discuss the results of our efforts, related work, and potential future work.

2 OMOS AND LINKER TECHNOLOGY

Separate compilation of program sources typically results in the generation of multiple *object files* which contain the generated program code and data. A linker is the program responsible for combining the object files and resolving inter-object file references. The linker manages large-grain code placement within an executable image. The decisions the linker makes with respect to code placement, in conjunction with the granularity of its data, determine whether a procedure is likely to be placed on the same page as the procedures it references. As program sizes increase, linker placement policies have an increasing effect on working set size and virtual memory utilization. In this paper, we are particularly concerned with the Unix linker. This linker is widely used, and while some of its shortcomings are particular to Unix, most of its problems are present in all linkers.

The first problem commonly encountered with linker policies concerns the granularity with which names are clustered. In an ideal system, if one were to reference a symbol A, the linker would locate and extract the code associated with the symbol, then iteratively extract only other symbols referenced by that code. This ideal is difficult to achieve because most linkers work at the object file level, and extracting symbol A means including all other symbols and associated references found within that object file, including but not restricted to those required by A.

Well-organized source files, compiled and carefully grouped into libraries of object files, come close to achieving the ideal of allowing a partial ordering of symbol references. More typically, the organization of object files reflects what is convenient for the programmer; the entities found in a relocatable executable are usually related, but often the relation is at the conceptual level, rather than at the symbol level. Clearly, if more than one procedure is exported from an object file, there exists the possibility of including code not explicitly referenced in the resulting executable (along with all the code *it* references). As software changes over time, the chances of grouping non-interdependent procedures within a single object file increase.

Another problem is that current linkers are controlled by the programmer. Linkers obey a series of programmer commands indicating in what order to bind object files. The object files bound together consist of either explicitly specified files, or selections made implicitly from libraries of object files. In general, the order in which plain (non-library) object files are processed by the linker has no effect on the correctness of symbol resolution or program construction. Therefore, programmers tend to pay little attention to this ordering.

The usual implementation of object file libraries further worsens locality. To save time, linkers commonly process libraries in one pass. This means that the programmer must either arrange to reference all symbols that are to be extracted from the library prior to its processing, or explicitly process the library more than once. A side effect of this style of library processing is that library elements are extracted *breadth-first*. All procedures extracted from a library are processed and made physically adjacent in the resulting executable before the linker processes subsequent libraries or object files. As a result, there is very little chance that a procedure in one library will be physically adjacent to any procedures it may reference outside the library. We will see empirical evidence of this, as well as the fact that adhering to depth first call-chain order produces much smaller working set sizes in large programs.

Finally, a Unix-specific library problem has to do with the processing of *common* data definitions. Global variables in the C programming language are examples of common data items. C global variables may be defined (e.g., `int foo;`) as often as desired, as long as they are assigned an initial value only once. Static storage is allocated for the variable and all common definitions are mapped to the same location when the program is linked. A pure reference to a global variable (e.g., `extern int foo;`) does not produce a definition.

Difficulty occurs when a common variable definition is repeated in more than one library element. When the variable is referenced, the linker chooses one of the definitions — typically the first encountered — and binds the object file in which it is found into the program. If a programmer has defined common storage for a symbol in a library header file instead of declaring pure references, the effect can easily be that many otherwise unrelated elements define the common variable. In these cases, a random and completely unrelated object file, and all other object files it may reference, may be linked into the program.

Clearly, these problems are not significant when using relatively small programs and small numbers of carefully designed libraries. The issue of locality of reference has been given attention in the past, when system memory sizes were small and penalties for non-local references were high[14, 10, 7]. Even though machine address space and main memory sizes have been growing, applications sizes have grown proportionally. For contemporary applications such as X window system clients, whose code sizes are an order of magnitude greater than those of simple applications such as `ls`, the problem of non-local references to procedures is again significant. In addition, poor locality of reference puts a burden on other parts of the memory hierarchy, such as the TLB and cache. As processor and cache speeds increase, so will the relative penalties for TLB, cache, and page faults. Given that applications will continue to grow in size, there is a strong need to maintain good locality characteristics within programs.

One solution to the problem of poor locality is to use a procedure ordering that more closely serves the needs of the program, rather than the convenience of the program development environment. For best results, the ordering should take advantage of temporal information, as well as simple dependency information. Also, the ordering should be done automatically, so that it becomes a standard, transparent compiler optimization, rather than an inconvenient-to-use separate analysis and transformation procedure.

These goals are achieved by the OMOS object server, which provides a rich and flexible framework for manipulating objects and programs. OMOS constructs object instances dynamically, providing the system with the ability to transpar-

ently provide users with different (and, typically, equivalent) implementations for a given object. To achieve a better code order within user executables, we have implemented *monitoring* and *reordering* within the OMOS framework. Because we implement the Unix program loading facility (`exec`) using OMOS primitives, reordering extends transparently and seamlessly to user programs.

3 SERVER ARCHITECTURE

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 objects may embody familiar services such as `ls` or `emacs`, or they may be simpler "building-block" objects such as hash tables, AVL trees, etc. Meta-objects are templates describing the construction and characteristics of objects. Meta-objects contain a class description of their target objects.

OMOS permits clients to create their own meta-objects, or to load instances of meta-objects into their address space. For example, given a meta-object for `ls`, OMOS can create an `ls` instance for a client. Instantiating an object subsumes linking and loading a program in a more traditional environment. OMOS is designed to support clients running on a variety of operating systems, including microkernels such as Mach[1] or Chorus[19], or traditional monolithic kernels that have remote mapping and IPC facilities.

Meta-objects contain a specification, known as a *blueprint*, which includes the rules used to combine objects and other meta-objects to produce an instance of the meta-object. These rules map into a graph of construction operations, the *m-graph*, with each node representing one operation.

Nodes in the m-graph define operations used to generate and modify objects. These operations consist of *module operations* as defined in Bracha and Lindstrom[3]. Conceptually, a module is a naming scope. Module operations operate on and modify the symbol bindings in modules; module operations take modules as input and generate modules as output. The modifications of these bindings define the inheritance relationships between the component objects.

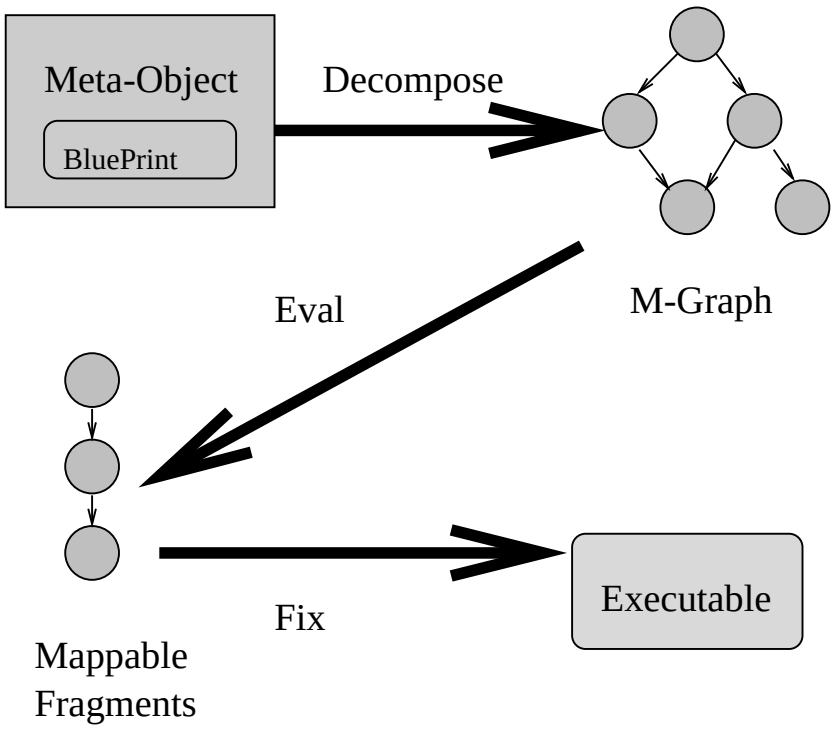

Figure 1 Operations for converting Meta-objects to executable programs.

The m-graph may also include some other non-module operations, such as operations that produce modules from source input, produce diagnostic output, group other operations into lists, etc. The set of graph operations into which a blueprint may be translated is described in more detail in Section 3.2.

In general, when OMOS receives a request for an instance of an object it must instantiate the object from a meta-object. To do this, OMOS compiles the meta-object into an m-graph. OMOS executes the m-graph, whose operations may compile source code, translate symbols, and combine and relocate fragments. M-graph operations may take other m-graphs as operands. Ultimately, the execution of the m-graph is resolved to a list of nodes which represent a set of mappable executable fragments. These fragments are mapped into the requesting client's address space. This sequence is depicted in Figure 1.

By using OMOS to produce the implementation of an executable given its name, we provide a level of indirection not found in traditional operating systems. This level of indirection takes advantage of the often overlooked fact that the executable file is really a persistent cached representation of a set of machine instructions. As long as that representation is a faithful reproduction of the source from which it was derived, a variety of transformations may be used to manipulate it. Since OMOS is an active entity, it can perform different transformations as conditions require.

3.2 Server Classes

OMOS is constructed from a set of classes which provide basic naming, class construction, and instantiation services. OMOS exports a hierarchical name space whose elements reference these server classes. Fragments represent executable code and data. They are the concrete implementation of modules and are typically implemented using the native executable format of the target system (e.g., *a.out*). Fragments export and import their interface through symbol definitions and references. Symbols in a fragment may already be bound to a value or they may yet be unresolved.

Meta-objects are central to OMOS. A meta-object describes the construction process used to instantiate an object. A meta-object supports three primary methods to create an object: `decompose`, `eval` and `fix` (see Figure 1). The `decompose` operation recursively builds the m-graph from blueprint information. The `eval` operation executes the m-graph, producing a set of modules as a result. The `fix` operation constructs a set of mappable fragments from the evaluated m-graph applying traditional relocations in the process. The result of the `fix` operation is cached by the meta-object for future use — subsequent operations may avoid constructing and executing the m-graph if there exists an appropriate cached version.

A blueprint lists the set of operations used to transform a collection of meta-objects and fragments into a set of mappable fragments. Currently the specification language used by OMOS has a simple Lisp-like syntax.

M-graphs are composed of nodes which are graph operators, meta-objects and fragments. The complete set of graph operators defined in OMOS is described in [16]. 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.

Rename: systematically changes names in the operand symbol table, and works on either symbol references, symbol definitions, or both.

Restrict: deletes any definition of the symbol and unbinds any existing references to it.

Copy_as: makes a copy of a symbol under a new name.

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

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

Source: produces a fragment from a source object.

Most of these operators have modules as operands and return modules as results.

4 OMOS PROGRAM MONITORING

We use the flexible nature of OMOS' object framework to implement a transparent program monitoring and optimization facility. To do this, a user (a system manager, most likely) indicates by name which meta-object is to be monitored. When instantiated, the resulting object includes interposed monitor procedures. The monitor procedures send an event trace back to OMOS, which analyzes this information to derive a desired ordering of procedures within the executable. Then OMOS reorders the base executable; subsequent instantiations use the new, optimized version. An overview of this process is depicted in Figure 2.

4.1 Monitored Object Setup

The first step in this process involves physically replacing the meta-object with a derived monitor class that overrides certain of the meta-object's procedures.

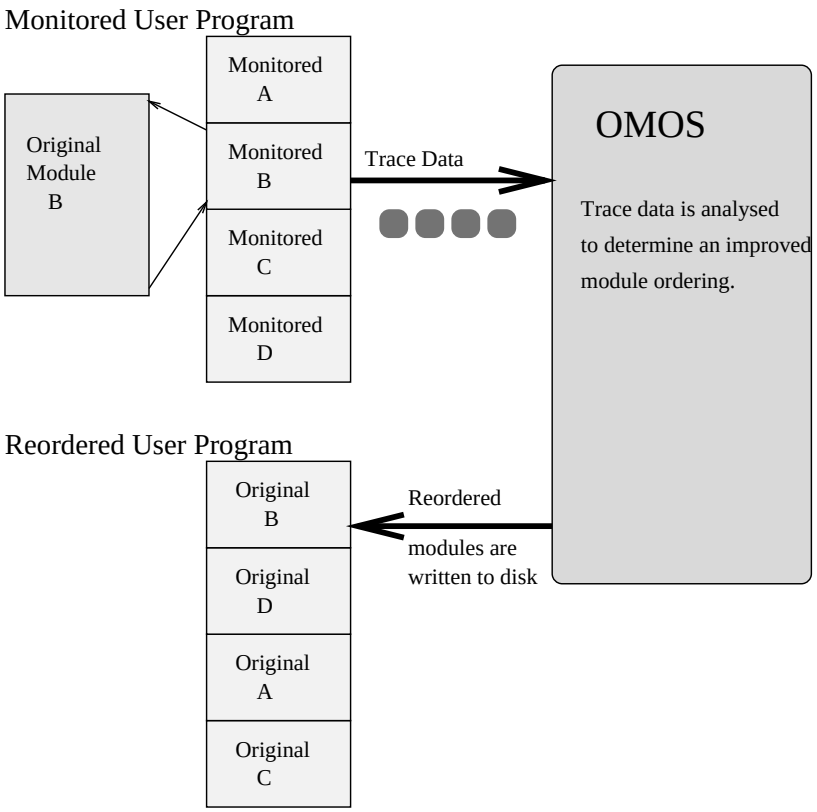

Figure 2 Overview of the monitoring and reordering process.

The privileged server method `monitor` takes the path name of a target meta-object and constructs the derived meta-object whose blueprint is a copy of the original blueprint. OMOS replaces the target with the new, monitor meta-object. Subsequent invocations of the target meta-object will dispatch to methods overridden by the monitor meta-object which will enact the monitoring and reordering functions.

The monitor meta-object performs the bulk of its work when the `decompose` method is first invoked. Recall, the `decompose` method generates the m-graph, the execution of which ultimately creates a set of mappable fragments comprising the code and data that make up the object. The first time `decompose` is invoked on the monitored meta-object, it invokes a `decompose` method on its base class to extract an initial m-graph. It then recurses through the graph, finding all of the fragments contained within. It rebuilds the graph, prepending a special `monitor` graph operation to each fragment.

During execution of the m-graph the `monitor` operation analyzes its operand, extracting the name of each procedure entry point in the module. The `monitor` operation generates an assembly source file containing a monitor stub procedure, or *wrapper*, for each entry point. Each wrapper exports an entry point with the same name as the original procedure. A `copy_as` operation is executed on the fragment, duplicating each entry point name as an internal name. This internal name will be referenced by the wrapper. A `restrict` operation removes the original name from the operand symbol table and breaks any existing intra-module bindings to it. The wrappers are compiled and `merge`d (i.e., linked) with the operand, generating a new fragment. A `hide` operation is invoked on the result to eliminate the intermediate names produced by the `copy_as` operation. Thus, the wrapper is transparently interposed between the caller of each procedure and the procedure itself.

Finally, a special version of `_exit` that knows how to perform a final clean up on the monitor state is interposed between the client and the system `_exit` routine. This result is linked with a library of monitor routines containing the support procedures which are invoked by the wrapper functions.

4.2 Monitored Object Execution

After the `fix` method has been invoked on the monitored object, the monitor code is in place and ready to generate log data. Each procedure wrapper logs information about entry and exit to the procedure. When an instance of the derived meta-object is mapped into a user program, the rest of the monitoring infrastructure is constructed: a thread is started in the server to collect log data which are returned from the monitored program via a communication channel.

On each invocation of a monitored procedure in the target process, the wrapper makes an entry in a log buffer local to that process. In order to preserve a valid stack frame, the wrapper replaces the return address on the stack with the address of an internal wrapper exit sequence. The wrapper saves the real return address on a private stack and transfers control to the monitored procedure. On exit from the monitored procedure, control is passed to the wrapper exit sequence; an entry is made in the log buffer, the real return address is retrieved from the internal stack, and control is returned to the caller.

When the log buffer is full, its contents are written over the communication channel. The monitor thread within OMOS collects and stores the contents in a file. The monitor version of the procedure `_exit` flushes any remaining log

information, signals a logical end of file to the server, shuts down the communication channel in the target process, and invokes the system `_exit` procedure to terminate the program.

4.3 Event Data Analysis

Once log data have been collected, OMOS runs an external analysis program to construct a dynamic call graph of the program from the event log file. The dynamic call graph constructed by the analysis program has a node for each instance of a procedure that is called, and an arc from the caller to the callee. The outgoing arcs are ordered temporally. Recursion is detected and converted to a cyclic arc.

A number of different reordering strategies can be applied to the log data. The analysis techniques produce an ordered list of procedure names. The ordering represents the order in which the procedures should be placed in physical memory to improve inter-procedure locality of reference. After an order has been generated via analysis, OMOS uses the list to reorder the fragments, as described in Section 6. The reordered version of the program will be used on subsequent invocations.

5 REORDERING STRATEGIES

The goal of the reordering strategies is to improve locality of reference. In general, the strategies we follow adhere to call graph order at the granularity of a procedure, rather than at the granularity of a relocatable executable file which the standard linker uses. The strategies discussed below are summarized in Table 1.

The first approach we take is to reorder based on a static call graph analysis. An analysis is made of call graph order; a given procedure is placed next to its first caller, as defined by the load order. Static analysis has the drawbacks that it may be difficult to do a proper call graph analysis if procedures are passed as arguments, and that there is no notion of how often or in what order procedures are called. Using profiling information to derive a call graph would provide a better idea of call frequency, but still lacks ordering information. In the following analysis techniques we use dynamic trace information to generate call graphs.

Strategy	Advantages	Disadvantages
Static call graph analysis	Requires only symbolic information contained in object files	Does not handle function pointers well. Uses no dynamic information such as frequency or order of invocation.
(1) Group singletons, rest by call-chain order	Requires only symbolic information. Isolates some initialization code.	Does not greatly affect main processing loop. May not isolate all init code.
(2) Group main processing loop in call-chain order	Separates main processing loop from initialization code	Currently requires the user to specify these functions
(3) Group habituals, rest by call-chain order	Identifies repeated calls on shared functions	May increase working set slightly
(4) Frequency of reference	Coaleses functions along most traveled paths	Identifying these paths is difficult

Table 1 Summary of reordering strategies.

Dynamic reordering strategies use the information collected from monitored program execution to guide the reordering. The first dynamic reordering strategy we apply initially separates out *singletons* — procedures that are only called once. Separating out singletons, divides the world into the set of commonly called procedures and the set of procedures that are used only once (and thus, will not be responsible for repeated page faults). We then order the remaining procedures using the dynamic call graph order. A given routine is placed next to its first caller. The call graph is traversed in a depth-first fashion. This strategy tends to split out initialization procedures and build up commonly used paths.

Experiments indicate that depth-first order produces better performance than breadth-first. The use of small procedures often causes programs to execute in a depth-first fashion. Breadth-first ordering might be applicable for programs having a small number of medium-sized routines. As we will see, typical procedure sizes in the types of applications we studied are quite small.

The second dynamic strategy involves having the user explicitly specify which procedure constitutes the beginning of the central processing loop. This specification separates the program into two distinct phases: an initialization phase and a main processing phase. The main loop is grouped in call graph order, followed by the set of initialization procedures. This strategy results in procedures common to both the main loop and the initialization procedures being grouped with the main loop, where, over time, they will tend to be called more often. Ultimately, we plan to use temporal information to derive program phases automatically.

The third dynamic strategy also uses a call-chain order, but first splits out *habituals* — procedures called frequently from a number of places — into a separate set of pages. The problem with habituals, such as `bcopy` or the string manipulation routines, is that they may be called often, and from a number of different sources. Placing them with any one call chain may unfairly make resident the rest of the procedures in that chain. To solve this, we cluster a number of the most frequently referenced procedures in the program by selecting a percentage of the total number of procedures. These procedures would also be prime candidates for cloning[9], which we intend to investigate as an enhancement in the future.

The fourth dynamic strategy involves ordering the call chain by frequency of reference, rather than in a simple first-called, depth-first fashion. At each level of the call chain, the most often traversed routine is chosen as the routine to be placed next in the output module and is next visited in the recursive traversal of the call graph. This strategy has the advantage that it will place together procedures in heavily traveled paths.

A difficulty with this strategy is that the out degree of any given node (the count of other nodes that node references) may not be a fair measure of the activity on that path; a node with a small out degree may still be accessed frequently because a large amount of activity is found beneath it. A call to a given procedure will result in touching the page of the callee on invocation *and* touching the page of the caller on return. Procedures that make many invocations may be as heavily "used" as procedures that are invoked many times. To take advantage of this knowledge, we perform weighting, wherein the weight of a node is calculated as a function of the number of times it is called *and* the weights of its children.

Clearly, different strategies are applicable for different programs or even different runs of the same program. Use of shared libraries increases the complexity of reordering by increasing the number of disparate uses of a given procedure.

In general, there is no optimal strategy for reordering all programs. We find, however, that usage information can provide orderings that are superior to those arrived at through static mechanisms. We demonstrate some of the particular strengths and weaknesses of these different techniques in Section 7, where we examine actual reordering results.

6 FRAGMENT REORDERING

The reordering transformation of a fragment must result in a new executable that is equivalent in function to the original. In principle, the transformation is simple:

1. Find the starting and ending offsets of all procedures in the executable code.
2. For each of the procedures in step 1, find all the relocations that are applicable to the procedure and all symbols that are defined within the procedure offset range.
3. For each of the procedures in step 1, move the procedure contents, adjust the symbol values of symbols defined within the procedure offset range, and adjust the offsets of the relocations applicable to the procedure.

In practice, optimizations performed by the compiler, linker, and assembler complicate the transformation. For example, a common compiler optimization puts constant data (e.g., strings) in the same segment with executable code. This makes location of the end of a procedure more complex. If the constant data are moved with the procedure, other procedures referencing the constant data no longer reference the correct addresses. Furthermore, if the constant data are referenced via a small pc-relative displacement, and the constant data are moved, after the move the displacement is wrong in all instructions accessing the constant data. Worse, the new displacement could exceed the reach of the instruction.

Another problem results from the assembler and linker performing optimizations to reduce the number of relocations that need to be performed during later steps. For example, references to defined symbols can be relocated by the assembler or linker. If the relocation is performed and the procedure is later moved, the original relocation becomes invalid. To allow object file reordering,

no relocations may be performed until the reordering has been accomplished. We have modified versions of the GNU assembler and linker to inhibit these troublesome behaviors.

7 THE RESULTS

We tested the OMOS reordering facilities using a version of OMOS which runs under the Mach 3.0 operating system, using a single server implementation of BSD 4.3 Unix. The machine was an 25 MHz Intel 80386 with 32 kB of cache and 16 MB RAM. We used the X program `xmh` as a test case, since it is constructed using layers of modules taken from several different libraries. The average length of `xmh` routines is 241 bytes. The binary is 405 Kb of text and 551 Kb total. Mach/Unix invokes it as a *demand-paged* binary, i.e., faulting in each page as it is first referenced.

In order to produce consistent results, we created special versions of the procedures that an X application uses to receive X input events. These can either make a record of the incoming events in a file, or retrieve events from a file and simulate their occurrence. The retrieval mode allows us to "play back" an earlier session with an X application. We also created a version of the procedure `_exit` which would report the number of page faults a program generated during its execution, since the Mach Unix server does not provide that information to the *time* utility. We interposed these procedures in the application using OMOS facilities, recorded a 10 minute `xmh` session, then replayed that session on a quiescent system under a number of different conditions to obtain our performance figures in multiple runs.

We tested six different strategies: a control with no optimization, a test of static call graph analysis, and the four dynamic strategies described in Section 5. We changed the amount of memory available to the system by wiring down free pages, depriving the application of available memory and forcing it to generate page faults. We observed the effect this deprivation had on the application's execution time and page fault rate. Figures 3 and 4 show the increase in execution time as available memory decreases. A graph of page faults versus available memory traces out a near-identical set of curves, demonstrating the increasing domination of page fault time as the amount of available memory decreases.

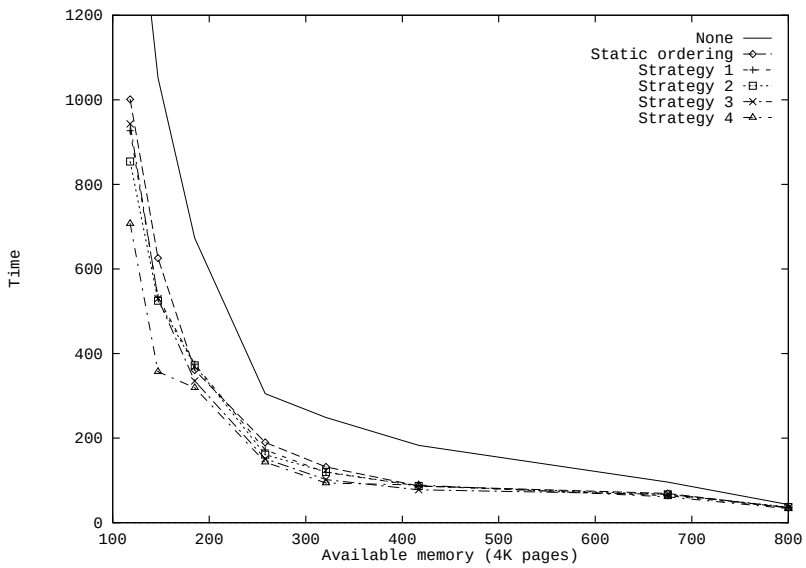

Figure 3 Time (seconds) versus available memory.

Turning to Table 2, we notice from the last column, where the application is given as much memory as it can use, that dynamic reordering results in fewer page faults and a 23% speedup. This reduction in page faults is the result of grouping the code for procedures that are never called on their own separate pages. Ordinarily, this code could have fallen on the same page as an active routine and been paged in due to a "false sharing" relationship. The decrease in execution time in this case can be accounted for, by improved cache utilization.

From the other columns, representing memory-short conditions with significant paging, we see that static reordering produces a substantial improvement in paging behavior, but that the more subtle improvements found in the more complex strategies prove to be significant as memory becomes scarce. Under these conditions, the dynamic methods give speedups ranging from 36–66%, while static ordering yields speedups from 28–52%.

Finally, we notice that the strategies of intermediate sophistication, such as strategies 2 and 3, actually perform slightly worse than the simpler policy of strategy 1, for some moderate values of available memory. This decline indicates that there is a cost to separating frequently called procedures from their callers; by putting them on a separate page, the working set is effectively

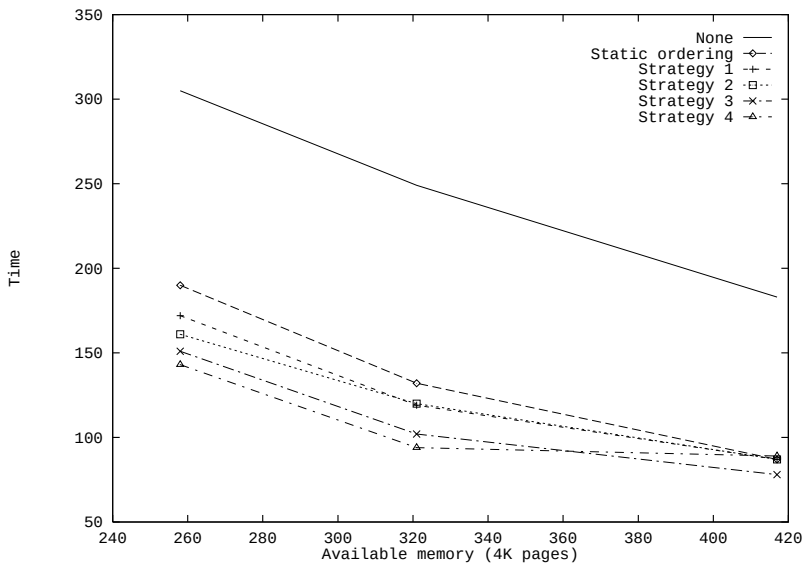

Figure 4 Blowup of time versus available memory.

Strategy	Available Memory (in 4k pages)							
	118	147	185	258	321	417	675	800
None	∞	1052	673	305	249	183	96	43
	∞	8755	4680	1623	1056	597	160	120
Static ordering	1001	626	361	190	132	87	69	35
	8209	5309	3024	975	608	225	121	120
Strategy 1	927	533	375	172	119	87	64	36
	8008	4578	2456	841	403	211	119	113
Strategy 2	854	525	372	161	120	87	67	37
	7334	4374	2310	768	540	234	125	113
Strategy 3	943	530	335	151	102	78	67	36
	8241	4190	2071	686	447	241	124	113
Strategy 4	707	357	319	143	94	89	61	33
	5979	2741	1926	674	318	247	116	113

Table 2 `xmh` Program Performance Data $\left(\frac{elapsed\ time\ in\ seconds}{page\ faults}\right)$.

increased by some near-constant amount. This expenditure becomes effective

as the rate of page faults increase and the value of accurate prediction of which pages are likely to be faulted on increases. This anomaly reinforces the need to investigate the use of code duplication for frequently used procedures.

8 RELATED WORK

There is a large body of work related to monitoring program execution to derive optimization information[8, 12]. Traditional techniques have often involved performing source to source translations, or modifying intermediate assembly code to insert the monitoring code. Laurus and Ball[13] use a technique of rewriting program executables and inserting monitoring code which is analogous to, but more ambitious than OMOS. Miller[4] uses the technique of *binary wrapping* to augment compiled procedure behavior, as OMOS does when generating monitored versions of programs.

A variety of work has been done on the problem of automatically improving locality of reference within programs in overlay systems and early paging systems[5, 14, 10, 7, 2]. Some of this work concentrates on instruction reference locality; others concentrate on data reference locality. These systems tended to organise programs as a collection of *segments*, each of which occupied some fraction of a page. Their work was based on reordering the segments to improve paging behavior.

Hatfield[10] and Babonneau[2] used an *adjacency matrix* to record inter-sectional references. Inter-sectional references were signalled by page faults, whose frequency was thought to approximate that of inter-sectional references. The system developed by Babonneau, et. al. makes reordering and monitoring a standard part of the linking process. Ferrari[7] used *critical working set* analysis to obtain adjacency mappings. The intent of the critical working set analysis is to observe inter-segment fault patterns as functions of time, and not absolute values. By positioning segments to minimize the number of cases where an inter-segment reference would fall outside of the process working set, the number of page faults is reduced.

More recent work focuses on the related problem of locality of reference within a cache[11, 6, 15]. McFarling[15] worked on reducing cache miss rates in direct-mapped caches. His algorithm allows a program to be restructured at the basic block level so that instructions that need to be in the cache at the same time are not mapped into the same cache line. McFarling studied a set of

10 benchmark programs using a range of cache sizes from 128 to 16384 words. They determined that regardless of the cache size, their algorithm is as effective as tripling the cache size.

Hartley[9] used procedure replication as a way to bound the locality of reference for a given point in the program. Pettis and Hansen[17] did work both to improve ordering of procedures and ordering of basic blocks; they concentrated more heavily on reordering basic blocks and used a single, straightforward algorithm for ordering procedures. They construct a weighted graph of basic blocks, which they successively refine, merging heavily weighted nodes together (mirroring what is to become their ultimate physical placement). They found 8–10% performance increases from better cache and TLB use, and their work is incorporated in Hewlett-Packard's current linker and `fdp` programs.

In terms of granularity, the work described in this paper falls somewhere in between the classic work done to improve paging and more recent work on improving cache behavior. By operating at the procedure level, we avoid much of the machine-dependent manipulations required when attempting to reorder basic blocks, while still working with a relatively small-grain entity. The 23% speedup we observed in tests where no paging was induced (beyond the initial demand load) verifies that significant cache behavior improvements were made by reordering, even at the granulatity of a procedure.

All of the reordering schemes we have seen are designed to be used in response to borderline conditions — applications which use the limit of available memory space or bandwidth. The popularity of schemes such as these rises and falls over time with changes in the costs of memory, memory access techniques, application complexity, hardware domain, and other factors. Changes in the limits of technology may alter the relative importance of this class of optimization, but its validity does not change. By automating locality of reference optimizations, we remove them from the category of special optimizations performed (and reinvented) only when applications reach the limits of technology. The relative benefit of these optimizations may rise and fall over time, but their general utility remains.

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.

9 FUTURE WORK

Many interesting problems remain to be addressed by OMOS. There is work to be done in the area of monitoring policy. We currently use the results of one run to determine what constitutes "typical use" of a program — the assumption being that the run will be specially tailored to be representative of typical use. We plan to investigate the policy issues of collecting and interpreting larger samples of data. We also plan on investigating the merit of duplicating the code of frequently used procedures, rather than trying to determine the best match for a procedure used heavily in several places. We will also look into the issues involved in reconciling diverse uses of a common piece of code, as in the case of shared libraries, where a single execution profile can not accurately represent the typical use of a set of procedures. And, we plan to develop policies whereby several instantiations of an OMOS meta-object — each tuned for a different use — can be made available to client applications.

Locality of data reference is arguably more important than locality of instruction reference, but is a less tractable problem, due to the difficulty of monitoring data references and due to the existence of dynamically allocated data. However, many numeric applications make heavy use of large arrays of static data. We plan on analyzing a set of such programs to assess the worth of reordering static data.

The extensible nature of OMOS, and its knowledge of everything from source file to execution traces, make it applicable to other kinds of optimizations requiring run-time data. OMOS could transparently implement the type of monitoring done by MIPS' `pixie` system, to optimize branch prediction[11]. 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.

There are a host of engineering issues to be addressed in OMOS: protection, consolidating OMOS servers in a network, implementing a virtual file system interface, and perhaps most important, policies for managing main memory and backing store.

10 CONCLUSION

Most current linking technology makes poor use of virtual memory by ignoring problems of locality of reference in large programs. This has adverse effects on total system throughput. OMOS, an extensible object/meta-object server, provides a framework for automatically improving the performance of programs through improved locality of reference. OMOS can transparently insert performance monitoring code in applications and gather data about a program's run-time behavior. Using this data, OMOS can derive an improved program layout and reorder executable code fragments to increase locality of reference. The most effective strategies for determining better fragment ordering are based on data available only from a run-time monitoring scheme. Significant performance improvements were gained from this approach.

Acknowledgements

We thank Robert Kessler and Gary Lindstrom for the time they have spent reviewing this work, Jeffrey Law for helping us make the Mach BSD Server do new and interesting things, and Bob Baron and Daniel Julin for providing key services and insights in times of need.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, June 9–13, 1986. Usenix Association.

[2] J.Y. Babonneau, M.S. Achard, G. Morisset, and M.B. Mounajjed. Automatic and general solution to the adaptation of programs in a paging environment. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 109–116, November 1971.

[3] 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.

[4] J. Cargille and Miller B. P. Binary wrapping: A technique for instrumenting object code. *ACM SIGPLAN Notices*, 27(6), June 1992.

[5] L. W. Comeau. A study of the effect of user program optimization in a paging system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Gatlinburg, Tenn., October 1967.

[6] Digital Equipment Corporation, Maynard, MA. *CHORD, CHORD2*, 1993.

[7] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(1):614–620, November 1974.

[8] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 13:671–685, 1983.

[9] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Trans on Software Engineering*, SE-14(11):1640–1644, 1988.

[10] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[11] J. L. Hennessy and Thomas R. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):342, July 1983.

[12] Amir Kishon, Paul Hudak, and Charles Counsel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 338–352, June 1991.

[13] James R. Laurus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report Technical Report 1083, The University of Wisconsin, March 1992.

[14] T. C. Lowe. Automatic segmentation of cyclic program structures based on connectivity and processor timing. *Communications of the ACM*, 13(1):3–9, January 1970.

[15] Scott McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[16] D. Orr and R. Mecklenburg. OMOS — an object server for program execution. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, Paris, France, September 1992. IEEE Computer Society.

[17] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990.

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

[19] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, December 1989.