An Approach to Genuine Dynamic Linking

W. Wilson Ho

Division of Computer Science, University of California, Davis, CA 95616, U.S.A.
Email: how@cs.ucdavis.edu

Ronald A. Olsson

Division of Computer Science, University of California, Davis, CA 95616, U.S.A.
Email: olsson@cs.ucdavis.edu

SUMMARY

This paper describes a new approach to dynamic link/unlink editing. The basis of this approach is a library of link editing functions that can add compiled object code to *or remove* such code from a process anytime during its execution. Loading modules, searching libraries, resolving external references, and allocating storage for global and static data structures are all performed at run time. This approach provides the efficiency of native machine code execution along with the flexibility to modify a program during its execution, thereby making many new applications possible. This paper also describes three sample applications of these dynamic link editing functions: program customization, incremental program development, and support for debugging and testing. A prototype of this approach is implemented under UNIX as a library package called *dld* for the C programming language and is available for VAX, Sun 3, and SPARCstation machines.

KEY WORDS Dynamic Linking
Incremental Program Development
Program Customization
Debugging and Testing
UNIX

INTRODUCTION

Many conventional operating systems—such as UNIX, DOS, and VMS—assume that programs are static entities in the sense that construction of a program is completed before its execution. A program's functionality, control structures, number of subroutines, and requirement on library functions are all well defined and do not change once the program begins execution. However, some programming languages, such as LISP and Prolog, take an alternative approach in which they allow new functions to be added during the execution of a program. The assumption that programs are static therefore makes it very difficult to translate these languages directly into native machine code. Instead, they are interpreted by a runtime support system or pseudo-machine, which runs more slowly than native code on a physical machine.

This paper presents an new approach to program construction that allows object modules to be dynamically **defined** or **redefined**, and **added** to or **removed** from a process during its execution. In other existing systems, object modules can at best be dynamically loaded but not removed. Using this approach, the functionalities provided by a program during its execution can therefore change with time or in response to the environment. Thus, this approach retains the efficiency of executing native machine code and adds the flexibility of modifying a program during its execution.

A dynamic link editor, called *dld*, implements this approach under the UNIX operating system. It integrates or removes object modules at run time. *Dld* differs from other dynamic linkers in that not only can object modules be added to but they can also be removed from an executing process. Furthermore, these modules do not even have to be known or exist when the execution begins. This paper describes several applications—program customization, incremental program development, and support for advanced debugging and testing features—that illustrate the usefulness of this dynamic linking approach.

The major cost of dynamic linking is the onetime overhead in reading object modules from disk. The processing time spent on link editing is actually very small. Once the modules have been linked, the executing process runs at nearly the same speed as the equivalent statically linked process. In fact, the only significant drawback of this dynamic linking approach is that its flexibility makes it susceptible to misuse. For example, a process might be corrupted by linking in erroneous code, or careless use of dynamic linking in privileged system programs might create security problems. A later section discusses these drawbacks and possible remedies in more detail.

BACKGROUND

This section presents some basic concepts of program compilation and link editing; it also describes the structures of relocatable object files and executable files in general. While the description of these structures and the example given in this section might not directly apply to all operating systems, the underlying principles are all similar. Readers already familiar with these concepts may proceed to the next section. Further details on these basic concepts can be found in Reference 1.

Most contemporary programming environments do not convert programs written in high-level programming languages directly into executable machine code. Typically, the source of a program is contained in one or more files, or ***source modules***, each of which contains definitions of functions and data structures. These modules are first compiled into ***object modules***, which are then combined together into a single file. This file, known as the ***executable file***, has a well-defined format understood by the operating system and can readily be turned into an executing process.

An object module is the machine code equivalent of its corresponding source module. It contains a ***text*** segment (machine code for functions) and a ***data*** segment (machine representations of global and static variables, string constants, etc.). Throughout this paper, the term *module* is used whenever there is no ambiguity or need to distinguish between a source module and an object module. When a source module is compiled, the compiler generates an ***object file***, which, in addition to the object module, contains global symbol definitions and information that enables this module to be relocated. This extra information is necessary because global symbols defined in a given module might be referenced by other modules, which need to know the location of these symbols. Furthermore, the compiler or assembler does not know the location in the address space where the module will be loaded when it is combined with other modules. Therefore, relocation information must be recorded with the object file. Related groups of object files of commonly used source modules—such as system services, input/output operations, and mathematical functions—are often combined into archive files, or ***library files***. As a result, users can specify a large number of related object files by giving only one file name.

The activity of loading and relocating object modules into an executable file is generally referred to as ***link editing*** or simply ***linking***. This activity maps each module to a section of the virtual address space, resolves global symbol references across modules, and allocates storage for the global data structures. Each module is then relocated accordingly and the results are written out to a file in the executable format.

Object modules to be linked together can be taken from individual object files or library files. A typical link editor usually handle these two types of files differently. For a simple object file, the object module it contains is always loaded into the executable file. For a library file, since not all modules it contains are generally needed, only those modules defining an unresolved external reference are loaded. Since a module from a library may itself contain references to other modules, loading it may generate additional external references. Therefore, link editors are responsible to search through the library files to ensure all required modules are loaded.

The ability for a linker to automatically select and load only the required modules from a library file is very important: it alleviates the need for a user to keep track of which modules are required. However, the executable file might still contain inaccessible functions or data because even if only one of the functions defined in a module is needed, the linker loads complete the module.

Figure 1 shows an example of linking 3 files: main.o, sub.o, and libc.a. Main.o and sub.o are object files and libc.a is a library archive, containing object files lib1.o, lib2.o, ..., libn.o. As shown in the figure, main.o contains references to sub.o and the function foo, which is defined in lib1.o. Function goo, also defined in lib1.o, contains a reference to hoo defined in lib2.o. Since the smallest unit of linking is an object module, lib1.o is loaded completely into the executable file, even though goo is actually not referenced at all. Furthermore, lib2.o is also loaded because it defines hoo, which is referenced by goo. The resultant executable file generated will contain 4 modules: main.o, sub.o, lib1.o, and lib2.o.

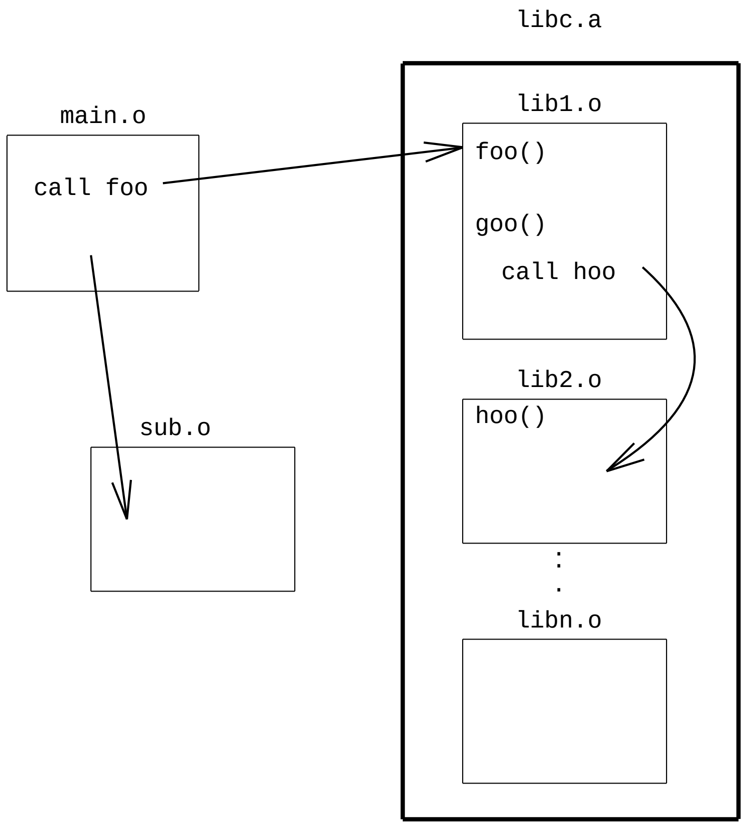

Figure 1: Linking of object and library files.

DYNAMIC VS. STATIC LINKING

Static linking and its limitations

Most operating systems assume ***static linking***. That is, the link editing step is carried out only once to produce an executable file, which is loaded directly into memory when it is executed. During the lifetime of an executing process, the (virtual) locations of the text and data segments cannot change. As a result, these operating systems can safely allocate the remaining address space for the stack and dynamic data storage area.

Systems that employ static linking require all global symbols to be well defined at link time. This requirement is a disadvantage because all object and library files must be available during the construction of an executable file. As a result, it is difficult to test and debug portions of a large program incrementally before the whole program is completely written.† Furthermore, the entire program has to be relinked if any of the object modules are modified, or if new modules are to be added. Relinking of all object modules can be very time consuming. But unfortunately with static linking, it is unavoidable even in situations where most of the object modules remain unchanged.

These drawbacks make static linking unsuitable for certain applications. For example, a graph plotting program may permit users to specify their own arithmetic functions to be plotted. It would be most convenient if users could define their own functions using the programming language with which they were most familiar, and then incorporate these new functions into the plotting program. However, the use of static linking precludes this obvious approach because the text segment of a program cannot be changed. Instead, most plotting programs incorporate their own special interpreted language in which users define their own functions. Consequently, users are forced to learn a new language, which might not be as powerful as a general purpose programming language, thus limiting the expressiveness of the users. Furthermore, the design and implementation of an interpreter for such a language increases the complexity and development cost of the originally very simple program. Efficiency of the program is also degraded because, in general, interpreted code executes more slowly than native machine code.

† The best a programmer can do is to write dummy routines for the unfinished part.

Dynamic linking

Unlike static linking, ***dynamic linking*** allows a process to add, remove, replace, or relocate object modules within its address space during its execution. In other words, programs are allowed to change. During the lifetime of its execution, a program may have new modules added, old modules removed, or even evolve into a completely different program. For compiled languages, the traditional concept that the code of a program does not change is no longer valid.

Some existing systems support what they call dynamic linking but it is actually load-time linking. For example, in SunOS version 4, link editing consists of two phases[2]. After the static link phase, the executable file created contains only references to, but not the actual code for the library routines. Integration of these routines with the executable file is carried out by the load-time link phase, during which a linker is called to search and load the missing routines from the library before control is passed to the main procedure. This system requires all global symbols to be declared in the library, though not necessarily defined, during the static link phase. After execution begins and control is passed to the main procedure, no modules can be added or removed. Furthermore, if any external symbol reference cannot be resolved, the program will be aborted. A similar approach is used by HP-UX[3], UNIX System V Release 4[4], and VMS[5]. In addition to load-time linking, OS/2[6] also supports run-time linking. But this scheme requires much operating system support and dynamically linked modules cannot reference any external symbols defined by other modules except the module entry point. Old systems such as MULTICS[7] and languages such as COBOL also support some kind of flexible linking/loading mechanism. However, none of the systems mentioned above allows a program to dynamically modify its functionality during the execution.

One incremental compilation system[8] uses a different approach: it provides library procedures that load additional object modules into the data area of a running process and allows these modules to be executed. This system does not relocate or resolve external symbol references. Instead, it locates every symbol indirectly through an on-line symbol table pointed to by a special reserved register, thus causing a degradation in execution speed. Furthermore, this system requires extensive modification to the compiler, assembler, and the libraries to accomplish this special addressing scheme. While this system allows modules to be added incrementally to an executing process, old modules cannot be removed. Consequently, using this system to implement applications such as interactive program development may not be practical because obsolete modules accumulate and may eventually fill the memory space.

DLD — A GENUINE DYNAMIC LINKER

This section describes the major design and implementation issues of a software package called *dld* that provides all the functionalities of a dynamic linking system. The current implementation† of *dld* is a collection of library routines that are called by programs written in C running under the UNIX operating system. No modification of the existing compiler or assembler is necessary. The standard UNIX system linker is used only to create the initial executable file. All the dynamic link editing is carried out by invoking routines provided by *dld*. Although *dld* is targeted for C and UNIX, the underlying concept—the use of library functions to build a dynamic linker—is applicable to many other programming languages and operating systems.

An overview of dld

Dld's two basic operations are ‘‘link’’ and ‘‘unlink’’. It also provides supporting functions for looking up the addresses of global symbols and entry points of functions.

The link operation is performed by the function dlink(char *filename), where filename specifies either a relocatable object file or an object library. If the specified file is a relocatable object file, it is completely loaded into memory. If it is a library file, only those modules defining an unresolved external reference are loaded. Since a module in the library may itself reference other routines in the library, loading it may generate more unresolved external references (as was seen in the example in figure 1). Therefore, a library file is searched repeatedly until a scan through all library members is made without having to load any new modules. Since a UNIX process cannot expand its text segment, dlink allocates storage for all these new modules from the dynamic data area—the *heap*—using malloc, UNIX's memory allocator (assuming the system has writable executable memory). After all modules are loaded, dlink resolves as many external references as possible. Note that some symbols might still be undefined at this stage, because the modules defining them have not yet been loaded.

† A number of functions in the current implementation of *dld* are borrowed and modified from the *ld* link editor developed by the GNU project of the Free Software Foundation.

Unlinking a module is simply the reverse of the link operation. The specified module is removed and the memory allocated to it is reclaimed. Additionally, resolution of external references must be undone. For instance, if the module foo is to be unlinked, all references to any symbols defined in foo from other parts of the program must be marked ‘‘undefined’’. *Dld* provides two functions for unlinking a module: unlink_by_file(char *filename, int hard) and unlink_by_symbol(char *symbol, int hard). The first function requires as a parameter the filename corresponding to a module previously linked in by dlink, while the second function unlinks the module that defines the specified symbol.

Figure 2 shows a simple example illustrating some of the *dld* functions. This program repeatedly reads from the standard input the name of an object file and that of the user function to be executed. It then links in the specified file, executes the named function, and finally removes it from the memory.

On line 13, the function dld_init(char *filename) performs the required initialization of the *dld* package. It takes as argument the initial executable file of the program and loads the symbol table information of this file into memory. Line 24 demonstrates how the entry point of a dynamically-linked function can be obtained. The value returned by get_func(char *func_name) can later be used as a pointer to the function, as shown in line 27. The predicate function function_executable_p(char *func_name) tells whether the specified function can be safely executed, i.e., whether the execution of this function might lead to referencing any undefined symbols. The precise definition of this predicate function is described later. In this example (line 29), unlink_by_symbol is used to remove the new module.

Semantics of unlink

As seen above, the unlink functions actually take two parameters. The first one is the name of a symbol or file, while the second one is a boolean parameter. When the parameter hard is zero (*soft unlink*), the specified module is marked as ***removable*** but it is actually removed from memory only if it is not referenced by any other modules. On the other hand, if the parameter hard is non-zero (*hard unlink*), this module is removed from memory unconditionally. Since unlinking a module may leave some remaining removable modules unreferenced, a garbage collector is always called to remove these

```
/* The name of the object file and the function name are read from the
   stdin.  The named function is invoked through a pointer.  For
   illustrative purpose the object file is unlinked right after the
   invocation.
*/
#include <stdio.h>
#include "dld.h"

main (int argc, char **argv) {

    char file_name[80], func_name[80];

    dld_init (argv[0]);

    printf ("object file? ");
    while (gets(file_name) != NULL) {
        register void (*func) ();

        dlink (file_name);

        printf ("function name? ");
        gets(func_name);

        func = (void (*) ()) get_func (func_name);

        if (function_executable_p (func_name))
            (*func) ();

        unlink_by_symbol (func_name, 1);

        printf ("object file? ");
    }
}
```

Figure 2: Simple illustration of the dld functions.

unreferenced modules. As a special case, library modules are always considered removable and are garbage-collected whenever they are not referenced by other modules.†

† In the next version of *dld*, users will be allowed to explicitly keep a library module in memory even if it is not referenced.

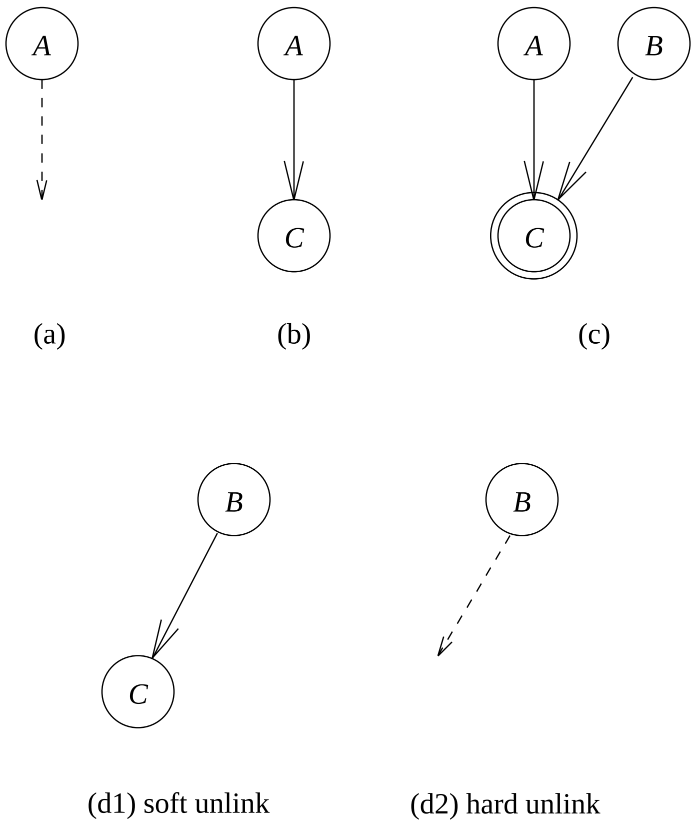

Figure 3: Illustration of the two different unlink options.

The semantics of these two unlinking options can best be explained by an example. Consider two modules, *A* and *B*, both containing a call to a function defined in *C*. These modules are linked and removed in the order as shown in figure 3. *A* is linked in first (a). Then *C* is also linked (b). When *B* is linked in, all its external symbol references are resolved successfully because *C* has already been loaded into memory. Thus, *C* does not need to be linked in again. Instead, it is shared by both *A* and *B* (c).

Now, suppose that *A* is unlinked, and in order to clean up any module referenced by *A*, *C* is being unlinked, too. Should *C*, which is currently referenced by *B*, be removed from memory? Both possible semantics are reasonable. If *unlinking* a module means it may be removed if it is not being referenced anymore, *C* should not be removed (d1) because it is still referenced by *B*. On the other hand, if *unlinking* literally means removing the specified module, *C* should be removed, resulting in *B*'s reference to *C* becoming unresolved (d2).

Applications might require either or both unlink options. Thus, *dld* supports both options and leaves the decision to the programmer. For example, if the module defining a commonly used function is to be replaced by an enhanced version, a hard unlink is desirable. A new version could then be linked in, and all references to the original function would be modified to point to this new function. On the other hand, some modules might be referenced from many different places in the program. It is not easy for a programmer to keep track whether a module is no longer referenced by others and thus safe to be removed from memory. The programmer might even be unaware of some such references. For example, functions such as strlen and malloc are referenced by at least a dozen other library functions in the C library of BSD UNIX. Thus, these functions often need to be kept in memory even if the main program does not invoke them explicitly. In these cases, it is desirable to use soft unlink so that the specified module is removed only when it is no longer needed.

Implementation of unlink

The implementation of the unlink functions is built around a simple garbage collector. For each module *x*, a reference count records the number of modules that reference any symbol defined in *x*. If *x* is explicitly linked in by dlink (as opposed to implicit linking in the case for library modules), it is considered being referenced by the process itself, and thus its reference count is at least one. A soft unlink of *x* causes its reference count to be decremented by one, while a hard unlink simply resets the count to zero. Furthermore, if *x*'s reference count becomes zero, the reference counts of all modules that *x* references are also decremented by one. When all reference counts have been updated, modules whose reference counts are zero are actually removed by the garbage collector.

In addition to the garbage collector, *dld* maintains data structures to hold information necessary to support unlinking. Corresponding to each module is a list of symbols that it references, a list of symbols that it defines, and a list of relocation instructions (known as an *nlist* in UNIX). This information is needed to update the online symbol table in order to reflect the disappearance of the unlinked modules from the program. When a module *y* is to be removed from memory, its reference list is used to locate those modules whose reference counts should be decremented. Symbols defined by *y* should be deleted from the symbol table. Other modules that reference any symbol defined by *y* now contain undefined references. These modules should be ‘‘reversely’’ patched so that instructions pointing to locations previously in *y*’s address space are reset. In short, all *dld*’s data structures have to be set to the state as if *y* had never been linked into the program.

Finally, side-effects—such as modification of global variables, input/output operations, and allocations of new memory blocks—caused by the execution of any function in a module are not reversed when the module is unlinked. If these side-effects need to be undone, they must be undone explicitly by the programmers.

Deciding if a function is executable

Since *dld* allows modules to be added to or removed from an executing process dynamically, some global symbols may not be defined. As a result, an invocation of a function might reference an undefined symbol. To solve this problem, *dld* provides a predicate function function_executable_p, which, as shown earlier in figure 2, takes as argument a function name and returns non-zero only if the named function is executable.

A function is *executable* if and only if all its external references have been fully resolved and all functions that it might call are executable. This recursive definition suggests that determining whether a given function is executable is non-trivial. Since each function might invoke other functions, the number of functions that can be reached from the original invocation can be very large. If any of these functions is not executable, the original invocation is also not executable. Consequently, determining if a function is executable could involve the examination of a large number of other functions. For example, the executable file for *gdb*, a popular UNIX debugger, contains 174 modules (object files) and 635 global functions. The efficiency of the algorithm† used to

† An optimal algorithm with respect to the order of complexity exists, but its detailed description is beyond the scope of this paper. Furthermore, the current implementation employs a sub-optimal but much simpler solution. This solution has a very small overhead and thus works faster than the optimal solution as long as the number of modules is not too large.

implement function_executable_p is thus of practical concern.

Several methods can be used to speed up this algorithm. First, the *executability* for each function could be actually calculated only once, and the results kept in a table. Subsequent calls to function_executable_p would then involve only a table lookup. This table would be invalidated whenever a link or unlink operation is performed, and would not be updated until the next call to function_executable_p. As a result, unnecessary recomputation of the table between consecutive link or unlink operations could be avoided. Second, when new modules are added or removed, minimal changes to the table of executable functions should be made. There is no need to rewrite the whole table. Only those functions affected by the changes should be examined. Lastly, although an algorithm that finds out the executability of a given function can be used repeatedly on every defined function, this method is in general very inefficient because a lot of the computation is duplicated. Instead, a different algorithm that finishes all the computation in one single pass could be used.

APPLICATIONS OF DLD

The flexibility provided by *dld* allows a new style in writing programs using compiled-based programming languages and makes many new kinds of applications possible. In particular, dynamic linking combines the efficiency of executing native machine code with the flexibility of runtime program modification. *Dld* is especially useful for highly interactive programs whose functionalities change in response to their user's input. This section describes several interesting applications of *dld*.

Program customization

Many sophisticated software packages usually allow some form of customization by the user. Depending on their personal preference, users can specify how these programs should interact with them. Usually these programs provide a group of different options from which users can select those that suit their needs. For example, in the UNIX editor *vi*[9] a user can specify whether the input text should be wrapped around automatically to the next line when input gets past a certain column. As another example, some versions of the UNIX command interpreter *csh* provide an option to log out the user automatically when it has been idle for a user-specified number of minutes.

This method of providing program customization is limited in the sense that users can only choose their preference from a set of predefined options. Whatever is not anticipated will not be available. For example, it is not possible to customize *vi* so that when it breaks up a long line, it also right-justifies the current line to a specific column. Likewise, it is not possible to tell *csh* to automatically log out at, say, 5:30pm simply because the need for this functionality was not anticipated. One possible solution is to let the users modify these programs according to their own preference and have separate private copies for each of them. Obviously, this method requires a lot of disk space and makes system software very hard to maintain and upgrade.

Dld provides a better solution by allowing users to add or remove new functions or modify the application programs to suit their own preference. The system then needs to keep only one copy of each program that is loaded with default options. If users are not satisfied with any of the options provided, they can then link in their own routines and tell the program to use these user-defined functions instead of the defaults. Referring to the previous *csh* example, if *dld* is used, a user could perform the following steps to customize the autologout feature:

- Use unlink_by_symbol to unlink the function that handles autologout.
- Modify the way the autologout function interprets its argument. For example, "17:30" means logout at 5:30pm while "17" means logout if *csh* has been idle for 17 minutes. Also make necessary adjustments so that the alarm clock is set correctly.
- Compile this new autologout function.
- Use dlink to link this function into *csh*.

Note that this method requires users to have some knowledge on the implementation of the application programs. Alternatively, the application programs can provide ‘‘hooks’’ or some well-defined, uniform interface to which the users can attach their own customized routines. Users should follow the specified convention when writing their own functions and the application programs may provide a special routine to handle the loading of the user-defined functions. This routine should hide from the users the details of invoking *dld* and of setting up the function entry points.

Incremental program development

Since *dld* does not require all source modules to be available when creating an executable file, it makes incremental program development possible. Programmers can start testing their programs before they finish writing the complete program without providing dummy definitions for the missing procedures. As long as the test input data does not cause the program to reference modules that have not yet been linked (i.e., does not invoke a function f such that function_executable_p(f) returns zero), the execution will proceed smoothly. Programmers can therefore develop programs incrementally and carry out thorough tests on individual modules before working on the next one. Furthermore, when new modules are added incrementally, the dynamic linker needs only to resolve external symbol references related to these new modules. In contrast, if static linking is used, link editing of the whole program has to be started anew. In other words, the use of dynamic linking can in general speed up the program development cycle.

Some systems[10, 11] support incremental program development by providing an interactive interpreter for the underlying programming language. These systems allow programs to be written, tested, and modified interactively. However, interpreted code usually runs more slowly than native machine code. The performance of an interpreter is often unacceptable when executing computationally intensive programs. With dynamic linking, interpreted and (possibly optimized) compiled code can be mixed together. Programmers might use compiled versions of the parts of the program that have been tested, while at the same time they can use interpreted versions of the modules on which they are currently working. The compiled modules are dynamically linked into the interpreter of these systems. These systems also provide the necessary interface for the interactions between the interpreted and compiled code.

As an example, suppose a graphical front end is to be added to an existing chess playing program, which is computationally intensive. During its development, code for this graphical front end is frequently modified, and so it is best run under an interactive interpreter. However, executing the chess playing routines under the interpreter would be unreasonably slow. With *dld*, compiled code for the chess program itself can be dynamically linked into the interpreter, and can be executed at the same speed as if it were statically linked. Thus, *dld* combines the ability of an interpreter to flexibly and completely control the execution of the still developing routines with the high execution efficiency achieved by the optimized machine code.

Support for debugging and testing

A dynamic linker also makes possible the implementation of many useful debugging features. A debugger can be used as an interactive interface to control the reconstruction of the user program and the execution of the *inferior* process, i.e., the process executing the program being debugged. For example, a programmer can use a debugger to stop the inferior process and invoke *dld* to remove the erroneous routines, correct them, and then link them in again. New debugging routines can be linked into the inferior process on demand. This feature is particularly useful when the need for such debugging routines is not realized until the program has been executing for a long time and finally arrives at some state of critical importance for debugging.

Just as incremental program testing is made possible, *dld* allows unfinished fragments of a program to be debugged. For example, a programmer might want to debug and test the symbol table routines of a compiler before moving on to finish the rest of the program. S/he might set up a debugger for the program shown in figure 2, and specify that these functions are to be linked in and invoked. The programmer could then set breakpoints, examine or modify program variables, or single step through the execution of statements in these functions in the same way as debugging a complete program. Note that information such as line numbers or variable names of these functions would not be known by the debugger, and must be loaded explicitly if source-level debugging is desired. Incremental loading of symbol table information is supported by some state-of-the-art debuggers, such as Dalek[12] and *gdb*[13].

Another important debugging application for *dld* is to help speed up the interaction between the debugger and the inferior process. Under most operating systems, a debugger can only control the execution or modify the memory image of the inferior process through a protected system call, e.g., `ptrace` in UNIX. This system call usually involves a considerable amount of overhead. For example, each call to `ptrace` results in two context switches between the debugger and the inferior processes. However, with the help of dynamic linking, a debugger can inject the most frequently used debugging functions into the inferior process's address space, insert calls to these functions at appropriate locations in the inferior's code, and then allow the inferior process to execute without any intervention from the debugger. Control is passed back to the debugger only when necessary.

For example, consider a conditional breakpoint that is put into the inferior process. That is, the inferior process is allowed to execute and the breakpoint is effective only when some specified condition is satisfied. Evaluation of the condition might involve a large amount of data, e.g., checking if an array is sorted. Also, the inferior process might hit the breakpoint many times before the given condition becomes true. If all the required information is transferred via a system call to the debugger everytime the inferior process hits the breakpoint, the overhead involved would be very high. With *dld*, a conditional breakpoint can be implemented by inserting code for the evaluation of the condition into the inferior process. Only when the specified condition is satisfied will control be passed back to the debugger. This scheme can be achieved as follows: First, the debugging code is linked into the inferior process's address space. Second, a branch instruction is overwritten into the location of the breakpoint so that the debugging code is executed instead. Lastly, the machine instructions that were originally in this location are moved to the end of the debugging code. By cleverly rearranging and relocating a few machine instructions, the insertion of the branch to and from the debugging code becomes transparent to the original execution. Since many unnecessary context switches and data transfers between the debugger and the inferior are eliminated, the efficiency of interactive debugging is greatly improved.

An alternative way dynamic linking can be used to speed up the interaction between the debugger and the inferior process involves the use of shared memory[14]. A large block of memory is allocated and shared between the debugger and the inferior process. Then the program to be debugged is dynamically linked and loaded into this shared memory. Control of the inferior process is then passed to the main procedure of this program. As a result, any update of data made by this program is readily observed by the debugger without any system call. Also, instructions of this program can easily be modified because they are loaded in the shared memory, to which the debugger has direct write access.

DISCUSSION

The ability to dynamically link and unlink object modules from an executing process provides much flexibility in the construction of a program. This ability allows the building blocks of a program to change or evolve with time: it turns a ‘‘program’’ into a dynamic entity. The functionalities of such a program can be changed in response to its interactions with the environment. As a result, dynamic link editing makes possible many new kinds of applications that would otherwise be very hard to implement.

Systems with similar capabilities have been introduced before. For example, programs written in languages such as LISP or Prolog can load in and execute new functions at any time during their execution. However, with only static linking, these systems have to be built on top of an interpreter or a simulated machine, which always runs many times slower than real machines. Dynamic link editing, on the other hand, combines the high execution speed of native machine code with the flexibility of dynamically modifiable programs.

As a positive side effect, *dld* helps speed up the program development cycle by allowing incremental program construction and online maintenance. As the sample applications illustrate, when new modules are added to a program, only part of the program needs to be modified—the rest of the program need not be relinked. Incremental linking can save a considerable amount of time, especially for large programs that often take many minutes to link. Furthermore, the size of the executable files and the disk storage requirements can be reduced because the system can now keep only one copy for the commonly used library routines and have them dynamically linked into the executing process on demand.

Another potential optimization of storage requirement is to allow commonly used library routines to be shared among multiple processes. Many new versions of UNIX[2, 3, 4, 6] already support shared library. Although the current implementation of *dld* allocates storage for the dynamically linked library modules in the private data area of individual process, a simple modification would allow the process to use the shared copy instead.

The price for the flexibility provided by *dld* is perhaps the overhead in processing time for the link editing and the extra memory required for holding the symbol table and other bookkeeping information. In practice, the time spent in performing both the link and the unlink operations is very small when compared with that spent in executing the program itself. Once the linking is finished, the program can execute at the same speed as if it were statically linked. Although a dynamically linked function must be invoked indirectly through a pointer (as shown in figure 2), this indirection involves only one extra pointer reference. That extra cost is negligible compared with the cost of executing typically hundreds of machine instructions in the body of a function. In addition, preliminary studies show that the vast majority of the time spent in dlink is for reading the object files or libraries from disk. In other words, the overhead in resolving external symbol references and maintaining the symbol table is insignificant.

The current implementation of function_executable_p is not complete according to the given definition of executability. External references through pointers are not traced. That is, this function will still return non-zero if the named function uses a pointer to indirectly call another function that has already been unlinked. Furthermore, if one external reference of an object module is unresolved, all functions defined in this module are considered unexecutable. This approximation results in a very efficient implementation, albeit conservative.

Like many other powerful tools, there is always a danger that *dld* could be misused. Similar to its static counterpart, a dynamic linker simply combines object modules together and does not provide any extra protection on the existing code against corruption. For example, *dld* does not check if a function to be unlinked is still active (i.e., has a corresponding activation record on the stack). If such a function is unlinked, the executing process might crash when control is passed back to it because the memory originally holding the code for the function might have been garbage collected and reallocated. Also, a function that was executable at one point in time might not be so when other routines are removed. Therefore, the *executability* of a function should in general be verified everytime before it is called, or everytime after modules are added or removed.

Another potential hazard occurs when a dynamically linked routine is erroneous and its invocation could crash the executing process. Thus, programs that allow user customization by means of dynamic linking expose themselves to potential misuse or destruction. In particular, privileged system commands should never allow user modification for the same reason that privileged UNIX commands should not allow shell escapes[15]. Otherwise, an intruder can easily gain unauthorized access to system resources by modifying an existing privileged command. For other application programs, a well-defined, simple interface should be designed for users to hook-in their own functions.

CONCLUSION

This paper describes a concept of genuine dynamic link editing and the implementation of a working dynamic linker *dld*. This new approach of dynamic link editing allows users to add, remove, or modify compiled object modules of a program while it is being executed. As a result, application programs using dynamic linking enjoy both the efficiency of executing native machine code and the flexibility of modifying their functionalities in response to the changing environment. A large number of new applications, such as those presented, are thus made possible.

Dld has been implemented for VAX machines running Ultrix, and for SUN 3 and SPARC workstations running Sun Operating System version 3.4 or 4.0. It is a package of library functions callable by C programs; it requires no modification of existing software, such as the compiler and the assembler. Programs using *dld* suffer a slight overhead in loading new modules from disk and performing runtime link editing. Once these steps are completed, statements in the newly added routines execute at the same speed as if they were statically linked. *Dld* can be obtained free of charge from the authors.

ACKNOWLEDGMENT

Many thanks to Richard Stallman and others who made the GNU *ld* available to the public. A number of functions from *ld* are modified and used as the basis of the implementation of *dld*. Rick Crawford, Carole McNamee, Chris Wee, and the anonymous referees provided very useful comments on earlier drafts of this paper. Various people around the world who beta-tested *dld* have provided valuable feedback. Their help is sincerely appreciated.

References

1. L. L. Beck, *System Software: An Introduction to System Programming,* Addison-Wesley Pub. Co., Reading, Mass., 1985.
2. R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, ‘‘Shared Libraries in SunOS,’’ in *USENIX Conference Proceedings*, pp. 131-145, USENIX, Phoenix, AZ, Summer 1987.
3. M. Sabatella, ‘‘Issues in Shared Libraries Design,’’ in *USENIX Conference Proceedings*, pp. 11-23, USENIX, Anaheim, CA, June 1990.
4. AT&T, *ANSI C & Programming Support Tools,* UNIX System V Release 4 Programmer’s Guide, Prentice-Hall, Englewood Cliffs, NJ, 1990.
5. J. Hobbs, ‘‘Installed Shareable Images,’’ *Dec. Professional*, vol. 6, no. 4, pp. 78-82, April 1987.
6. G. Letwin, ‘‘Dynamic Linking in OS/2,’’ *Byte*, vol. 13, no. 4, pp. 273-280, April 1988.
7. R. C. Daley and J. B. Dennis, ‘‘Virtual Memory, Processes, and Sharing in MULTICS,’’ *Communications of the ACM*, vol. 11, no. 5, pp. 306-312, May 1968.
8. M. K. Crowe, ‘‘Dynamic Compilation in the Unix Environment,’’ *Software—Practice and Experience*, vol. 17, no. 7, pp. 455-467, July 1987.
9. W. Joy and M. Horton, *An Introduction to Display Editing with Vi,* University of California at Berkeley, Computer Science Division, May 1986.
10. S. Kaufer, R. Lopez, and S. Pratap, ‘‘Saber-C—An Interpreter-based Programming Environment for the C Language,’’ in *USENIX Conference Proceedings*, pp. 161-171, USENIX, San Francisco, Summer 1988.
11. B. B. Chase, ‘‘Selective Interpretation as a Technique for Debugging Computationally Intensive Programs,’’ *Proc. of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, pp. 113-124, St. Paul, MN, July 87.
12. R. A. Olsson, R. H. Crawford, and W. W. Ho, ‘‘Dalek: a GNU, improved programmable debugger,’’ in *USENIX Conference Proceedings*, pp. 221-231, USENIX, Anaheim, CA, June 1990.

13. R. M. Stallman, *GDB Manual (The GNU Source-Level Debugger), Third Edition*, Free Software Foundation, Cambridge, MA, Jan. 1989.

14. Z. Aral and I. Gertner, ‘‘High-Level Debugging in Parasight,’’ *ACM Workshop on Parallel and Distributed Debugging*, pp. 151-162, Madison, WI, May 1988.

15. P. H. Wood and S. G. Kochan, *UNIX System Security*, Hayden Book Co., Hasbrouck Heights, NJ, 1985.