DDL: Extending Dynamic Linking for Program Customization, Analysis, and Evolution

Sumant Tambe   Navin Vedagiri   Naoman Abbas   Jonathan E. Cook
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

While new software languages and environments have moved towards richer introspective and manipulatable runtime environments, there is still much traditional software that is compiled into platform-specific executables and runs in a context that does not easily offer such luxuries. Yet even in these environments, mechanisms such as dynamic link libraries do offer the potential of building more control over the deployed and running system, and can offer opportunities for supporting dynamic evolution of such systems. Indeed, the basic dynamic linking mechanisms can be exploited to offer a rich component-based execution environment.

In this paper, we present our initial explorations into building such support. Our approach is to extend the Gnu open-source dynamic loader to give the deployer control over the configuration of the system, and to be able to dynamically evolve that system. Applications of this capability include runtime component configuration, program evolution and version management, and runtime monitoring.

1 Introduction

Dynamic link libraries, also called shared libraries or shared objects, have the potential to offer a rich, dynamic, component-based deployment platform. Many, if not most, of the latest ideas in component-based software frameworks and development could be supported by the shared object platform. Private namespaces, naming service lookup for binding requests, interface checking, introspection and monitoring support, and dynamic reconfiguration are just some of the capabilities that could be supported—if the underlying framework enabled it to be.

Shared objects (or dynamic link libraries) delay the binding of externally needed symbols (functions, methods, global data) to the runtime of the program. An extra module, the dynamic linker, is loaded with the program, and accomplishes the dynamic linking necessary for the program to complete its execution. Since symbols are not yet bound, a dynamic linker could allow a tool builder access to the binding operation, and allow the tool to take a variety of actions. Inserting wrappers for tracing, security, assertion checks, and many other uses ought to be possible. Redirecting a binding to another symbol should be supported. Even run-time modification of bindings should be possible. All of this is feasible without any modification or translation of the object code of the application.

Current dynamic linkers do allow rudimentary control over the linking process. Most allow an application to *preload* a library, so that symbols in the preloaded library will take precedence over those in libraries loaded later. This is typically done for things like wrapping system calls with particular behavior (e.g., a virtual file system) or security checks. However, the preload mechanism is static and is cumbersome to program—the wrappers need to explicitly load and find the *actual* symbols that it is wrapping.

In this paper we describe DDL, a tool based on modifications and extensions to the dynamic linker that allow us to have dynamic control over the linking process, and to implement features such as those listed above. DDL allows the user to control the runtime configuration of the application, enables the easy construction of runtime monitoring tools and supports the runtime evolution of dynamically linked programs. DDL is a modification of the Gnu dynamic loader, which is part of the Gnu C library. Our current tests have only been on the Gnu/Linux/x86 platform, although the Gnu libraries (and the dynamic linker) are ported to many other platforms.

Section 2 gives a brief overview of dynamic linking. Section 3 discusses dynamic linking from an architectural perspective. Section 4 details our modifications to dynamic linking that allow link interception and runtime evolution. Section 6 describes how C++ interacts with the dynamic linking framework and how DDL can support it. Section 7.1 presents a brief look at one example tool that uses the DDL framework, while Section 7.2 presents some ad-hoc data analysis examples using DDL. Section 8 discusses lessons and ideas learned during the project. Section 9 presents related work, and finally Section 10 concludes with some ideas for the future directions that we are pursuing.

2 Dynamic Linking, Briefly

The fundamental action of a linker is to take multiple separately-compiled pieces of object code and resolve the unknown shared symbols into addresses, so that the object code can execute without any missing pieces of information. Dynamic linking leaves the symbol resolution process to be completed at runtime. The external symbols are still "resolved" during the static link process—however, only a placeholder dependency reference to the dynamic link library that contains the symbol is put into the executable object code.

From here on we will use the terms shared library and/or shared object rather than dynamic link library, because this term is more traditional, and it does capture the important notion that dynamically linked objects can be shared among processes. The code pages—which typically make up most of a shared library—are write-only and the code in them is compiled to be position-independent[1], and thus it can be mapped into multiple process spaces, even at different addresses.

In the Executable and Linking Format (ELF) [9], dynamic linking uses two tables: the Procedure Linkage Table (PLT) and the Global Offset Table (GOT). Calls to external functions (and often to internal functions as well) use these tables to effect an indirect call. Figure 1 shows a call to the *printf* function as it goes through the PLT on an already resolved entry. The actual call site in the program "calls" an entry in the PLT. The PLT is executable code, with basically a jump instruction for each entry. The jump is an indirect jump that uses the corresponding entry in the GOT—an address—to jump to the correct function. Before a symbol has been

[1]With the Gnu C/C++ compiler, generating position-independent code must be explicitly selected with the -fPIC option.

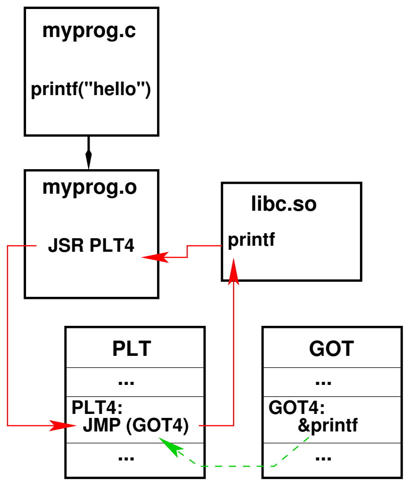


Figure 1: Function call through a dynamic link.

resolved, the GOT entry (in effect) has the address of the dynamic linker's symbol resolution routine in it.[2]

Thus, the PLT and GOT tables centralize the code that uses the dynamically bound links and the addresses of those links. The PLT is code while the GOT is data—a table of addresses of functions. The dynamic linker is invoked upon the first call to a function, and its job is to find the symbol (possibly needed to load the shared library into memory), determine its address, load that address into the correct GOT entry, and effect a jump to the function. The function returns directly to the original call site (since only jumps occurred in between), and all subsequent calls only incur a one-instruction overhead since the GOT now contains the correct address of the function.[3]

For position-independent code, symbols representing global data are also referred to through the GOT, but not through the PLT. The GOT is an address table for all external symbols (and internal globals, as well), while the PLT is specifically for function calls.

We briefly mention C++ here, but detailed expla-

[2]In detail, the PLT entry has a couple of instructions below its main jump. The GOT initially points back to these instructions. Their job is to push the symbol name as another parameter and then effect the jump to the dynamic linker, which still goes through yet another PLT entry in order to push the library name as a second parameter for the dynamic linker.

[3]Dynamic linkers do generally support binding modes other than first-call binding. Load-time binding allows resolution before the first call, essential for real-time systems, and suppressed binding forces every single function call to trigger a resolution (which never updates the GOT), which can be important in debugging situations.

nation of C++ is found in Section 6. The dynamic linking mechanism (and static linking as well) only knows about symbol names, it does not have inherent understanding of classes and such. When C++ is compiled, the compiler does *name mangling* to convert the class and method name into a single unique symbol. Because C++ allows method overloading (same name but different parameters), the types of the parameters are also used in the name mangling to produce a unique name for each method, overloaded or not, in a class. In this way, C++ is "invisible" to the dynamic linker, and class methods are only related in that their mangled names all include the same class name. A further complication is in polymorphic behavior, especially the mechanism of virtual methods that allow specific runtime-selected behavior based on the actual object type being used. A class *vtable*—a table of function pointers—is used to implement polymorphic method calls.

3 An Architectural Perspective

If one steps back for a moment away from the low-level idea of dynamic linking being symbol resolution, a broader picture of the meaning of what is happening can appear.

In using dynamic linking, an executable program is *incomplete*—it does not have the complete code to run. Instead, it needs external *services* to be able to run to completion. The dynamic linker's job is to find these services and connect the program to them so that it can use them.

In the abstract, the area of software architecture gives us a language to understand this interaction. The pieces of a system are called *components* and other pieces called *connectors* create the connections between the components [11]. We might be used to these ideas when thinking of, say, Java RMI and the stubs and skeletons acting as connectors for our remote method invocation, but the same basic idea applies to the dynamic linking environment.

Each shared object, including the application program, is a component that contains references to required services. These references are in the form of names (symbols). Additionally, each component also advertises its provided services, also referring to these by using names (symbols). There is no reason at all why the required service name must match the provided service name—indeed, it is constraints like these that cause global namespace pollution.

Rather, we can view the names as local specifiers of required and provided services. It is the architectural level that needs to provide a binding specification between component namespaces. This might be as simple as equating them (thus reducing the problem to normal linking), but it might involve translation of one name to another, binding a complex connector in between, binding to a remote service, or even more complex processes.

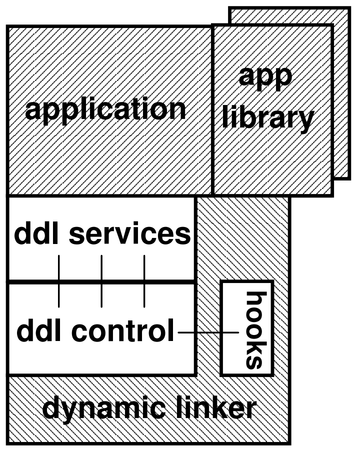


Figure 2: DDL system architecture.

This view is useful on many different levels. As already seen, it allows components to have independent namespaces, even for externally required or provided services. It allows an external framework to choose how to connect the components rather than having a single implicit mechanism. It allows the framework to choose to add monitoring or management capability into the system, by placing it in the connectors between application components. It enables interface translators to connect required and provided services that might have slightly different formats.

Understanding dynamic linking as just one mechanism for connecting independent components into a complete application, one which the default behavior of symbol matching is a throwback to a monolithic system, programming language centric viewpoint, enables us to place it alongside modern component system frameworks and to work to bring its implementation—the dynamic linker—up to date in its capabilities.

4 DDL: A Dynamic Dynamic Linker

While the long-term goal of our work is to move dynamic linking towards a full-fledged component framework, this paper presents just the beginnings of that work, and modified dynamic linker called DDL.

DDL is an extension to the Gnu dynamic linker, and is extensible itself. Figure 2 shows the high-level system architecture that DDL implements. The shaded portions indicate parts of the system that DDL does not modify. The application and application libraries are not modified, at the source or binary level, and the bulk of the system dynamic linker is unmodified.

At the lowest level, we added hooks into the dynamic linker itself so that we could interact with the linking process. On top of these hooks we built useful service abstractions so that tool builders would not need to start from scratch. Further, we implement a application services level that provides even higher level interaction for some types of common services—one such service is scripting language support.

5 Linker Modifications

One goal in this project was to have very minimal modification to the Gnu dynamic linker itself. We decided that any significant code we developed would sit outside of the linker. Thus, our essential modifications boil down to callback hooks in the linker code itself. The hooks themselves are described later; this section only describes the linker modifications.

To activate our modifications, we added linker code to check and use an environment variable, LD_REDIRECT. If this is not defined, our modifications are ignored, and the linker operates normally.

The callback hooks are in the form of function pointers. If LD_REDIRECT is defined, our code attempts to initialize the function pointers by doing symbol lookups—using the dynamic linker code itself. The symbols are of course not defined in the linker or in the application and its libraries. They are defined only *if* a tool library has been preloaded (using LD_PRELOAD), and thus the hooks internal to the linker get connected to tool functionality out in a library.

All of this happens once, at application startup. Only if the function pointer callback hooks are initialized properly will DDL act differently than the regular Gnu linker. If they are, then at key points in the linking process, the callbacks are invoked and the external tool has the opportunity to interact with and manipulate the linking process. In this, the hooks built into the dynamic linker do not provide an API to external services but rather they use an API provided by DDL control library. DDL control and the tools that use them are thus generally passive and event-driven.

5.1 Link Process Manipulation

The fundamental capability that DDL supports is link interception and redirection. This allows DDL and the tools that use it to peer into the dynamic linking process, and control it. The callback points are:

redirect_init : This is called once to initialize anything the redirection code needs. It is called before “main()”, so it should not depend on any initializations in the application.

redirect_isactive : This should return non-zero if redirection is desired. If it returns 0, no other redirection code is executed. This offers dynamic on/off control to the DDL services.

redirect_lookup : This hook is called before the linker tries to resolve a symbol. As parameters this function receives the symbol of the function to be resolved and the shared object from which this call is coming from (i.e., the main application or some shared library). It returns either a) the new name to which it should be redirected; b) the exact same pointer it received as a parameter, indicating no redirection is taking place; or c) a NULL pointer, also indicating no redirection.

redirect_definition : This function receives all the information about the symbol and link that have been resolved. For convenience it again provides the original symbol looked up and the name of the shared object the link request is coming from (as in redirect_lookup). It also provides the symbol that was actually resolved (different if redirect_lookup returned a different name), the library that the resolved symbol is in, the address of the resolved symbol, and the address of the GOT entry that is being updated. Through this, analysis tools can keep track of every link request and every symbol definition that is resolved.

redirect_offset : This function provides the offset (in bytes) that will be added to the address of the symbol that was used. This allows application-level PLT-style table-based redirection. It should return 0 if table-based redirection is not being used.

redirect_symdef : This function provides the information about load-time symbol definitions, many of which are not function symbols and so will not appear in the above callbacks. The

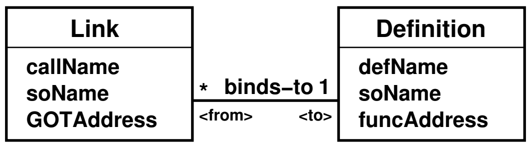


Figure 3: Link and definition UML.

symbol name, library name in which it occurs, and its resolved address are all provided.

Thus, with this interface, tools using DDL can inspect every link request, choose whether to redirect it to another symbol name, and record the information about the resolved symbol and about the link itself.

Our redirection capabilities allow for table-based redirection. That is, the redirection code can implement a mechanism similar to the builtin PLT-GOT table-based dynamic linking. The *redirect_offset* callback supports this capability. This will be explained in the next section in more detail.

Figure 3 shows how the **links** and **definitions** relate to each other and the information that uniquely describes each of them.

A link is defined by the original function name it is supposed to be linked to, the name of the shared object it is for (the main application or a shared library), and the address of the GOT entry where its binding address is stored. A definition is defined by its function name, the name of the shared object it exists in, and the address at which it exists. Further information that is useful to save is a reference from the link to its current definition, and in the reverse direction a set of links that currently reference a given definition.

Without using our redirection capabilities, all links will point to definitions of the same name. There are potentially multiple links to the same definition because each shared object that calls that function will have its own link (i.e., its own PLT/GOT and its own unique PLT/GOT entry for calls to that function). If our redirection capabilities are being used, then the called name associated with the link can be different from the defined name of the definition; thus it is important to keep track of these names separately.

5.2 Table-based redirection

In general, our link interception and redirection capability supports redirecting the calls of each *unique* function to some other *unique* function. While the mechanism does not prevent multiple redirections to the same symbol, in practice this does not make sense, outside of perhaps a few special cases. This is because the called function cannot differentiate between the calls, and thus cannot know which calls map to which original symbols. Thus, only if the arguments from all the calls were the same *and* the desired behavior was the same would it make sense to redirect multiple symbols to the same target.

There are cases in runtime monitoring and program maintenance, however, where it would be useful to have a *concentrator* function that did receive calls for multiple symbols and was able to differentiate them. For example, if one wanted to trace all calls in a program, it would be nice to have a single wrapper do the job rather than a unique wrapper for every function.

To support this functionality, we utilize the same basic mechanisms of the system PLT/GOT jump tables, but at our own user-defined level. The DDL interface that supports this is the capability of providing an *offset* to be added to the address of a symbol that is resolved.

To use this capability, we first must create a jump table. A simple (and non-thread-safe) example of this is below.

```
unsigned int func_id;

void wrapper_plt()
{
   asm("   movl $0, func_id\n\t\
           jmp wrapper\n\t\
           movl $1, func_id\n\t\
           jmp wrapper\n\t\
           ...
           movl $98, func_id\n\t\
           jmp wrapper\n\t\
           movl $99, func_id\n\t\
           jmp wrapper\n");
}
```

This example represents a 100-entry jump table, where each entry sets a global variable to its index value and then jumps to the wrapper function. Note that the program never *calls* the *wrapper_plt* function—rather, it jumps directly to one of the entries, which in turn jumps to the wrapper. Our use of DDL would redirect each symbol to $(wrapper\_plt + 3 + i * 15)$, where i is the entry assigned to that symbol. To do this it would do symbol redirection to “wrapper_plt”, allow the dynamic linker to find that symbol, and then add the offset using the *redirect_offset* DDL callback. At the same time, our DDL extension we would save the symbol string in a string table, at the same index being used in the jump table.

The wrapper function, using the global *func_id*, would have access to the index of the function currently being called, and from there could get the name of the function (since we saved it). After doing its tracing behavior (or whatever it is supposed to do), it could use *dlsym()* to resolve the original function and to call it. Functions with different argument vector lengths can still be handled by the same wrapper, since the reverse-calling convention ensures that extra argument data is ignored. The wrapper only needs to know the maximum argument bytes it needs to push on the stack.

While the jump table must be created in a platform-dependent manner, the basic idea remains essentially the same on most platforms, and through the symbol-plus-offset mechanism that DDL exposes to the user, effective use of a single site for multiple redirected symbols can be accomplished.

As one example, we used this capability to fully trace the SimpleScalar CPU simulator [3, 1].

5.3 Runtime Link Modification

In Section 5.1 we saw that we can maintain an internal data structure of resolved symbols, their address of GOT entries and the current definition of the function the link is referring to. Maintaining this information during the runtime of the program allows us to support dynamic program evolution through runtime link modification.

In order to modify a link, we simply need to change the address that is in its GOT entry to be the address of some other function. All the subsequent calls through that link will be directed to the new function. Note that these calls are from all the call sites in the shared object whose link we just modified. Thus, the granularity of program evolution is at the shared object level.

Once the link is initially resolved, however, the linker is never going to be called for that particular link again. Therefore, we have to have some way of regaining control over the execution of the program in order to perform runtime link modification. While in the long run some OS support would be needed for generic framework control of an application, as proof of concept we currently employ the OS's signal mechanism to accomplish this.

The user must first provide a specification file describing which links they want to modify. While the file does not need to be created when the application is first started, an agreed-upon file name and location *does* need to be known at startup, since there is no way to send such information through a signal. When a user-defined signal is sent to the application, our own handler reads in the specification file, traverses the internal data structures of links and definitions, and modifies the GOT entries for the specified links. When the handler is completed and the application regains control, subsequent calls on those links will go to the new definitions. The downside of the signal mechanism is that if the user process also installs its own signal handler for the same signal, our signal handler will likely never be called, since we installed ours first, before the application started up.[4]

In runtime link modification, it can be the case that a new definition for which links should point to may not yet have been loaded by the dynamic linker. This means that we must be able to not only search the existing definitions but also bring in new definitions, possibly even loading new shared libraries. We can do this using the standard *dlopen()* and *dlsym()* interface that already exists for user-level access to shared libraries.

Other work in dynamic program evolution has noted a desire to perform transactional updating—making sure that a module is not being actively used before updating references to or away from it [12]. This often boils down to checking the call stack to see if any functions in the module are active. We have not yet concerned ourselves with providing such capability, but plan to investigate these aspects in the future. In our current mechanism, existing calls through links being modified have already invoked the old definition, and those will eventually complete.

6 Supporting C++

The basic mechanisms used in static and dynamic linking were built to support simple symbol sharing—both data and functions—and they do not directly support more complex execution environments such as those that object-oriented programming languages like C++ demand. In this section we discuss using DDL on C++ applications, and concentrate on creating wrappers for C++ classes and methods, rather than just simply redirecting the method calls because creating wrappers exposes interesting problems and solutions.

When C++ is compiled, the compiler does *name mangling* to convert the class and method name into a single unique symbol, as discussed in Section 2.

[4]We believe we can use the DDL mechanisms themselves to wrap the signal() call and protect our signal handler while still giving the application its desired functionality, but we have not done so yet.

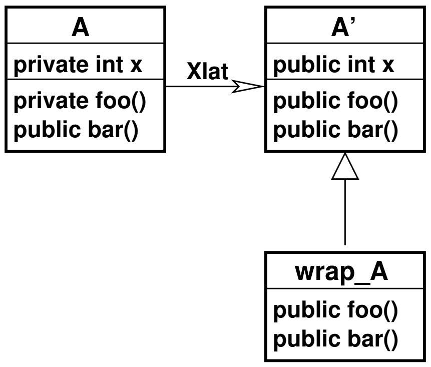


Figure 4: C++ wrapper definition.

It is possible to create wrappers in C for methods in a C++ class. Our redirection mechanism can map a request for the mangled symbol into the symbol for our C function. In C, one can directly call the mangled symbol as another C function, with an explicit *this* pointer as the first argument. The mechanism is even typesafe when compiled with a C++ compiler and using the *extern "C"* syntax. However, it is not very convenient. With the application coded in an OO methodology, we would like to create wrappers in the same way.

With wrappers in C++, our wrapper class(es) ought to mirror our application class(es). Moreover, the wrappers will need to know about (and invoke methods on) the application classes. The next subsections detail the methods for doing this.

6.1 Wrappers for C++ methods

To wrap any non-virtual method of a C++ class, we need to intercept the call to the mangled symbol of that method and redirect it to our own wrapper class' method. The approach we took to implement this is that we make the wrapper class to be a subclass of the target class (the class whose method(s) we want to wrap), and then define the wrapper method as the same name (and type signature) as the method which we want to wrap in our wrapper class.

Figure 4 shows the relationship between the application class A and the wrapper *classwrap_A*. In practice, we need to translate the declaration of A into a class declaration A' where all private members have been made public. We do not create an implementation of A'. This translation is done so that the wrapper class can have access to the data and methods. Note that A' is only used in compiling the wrapper, and we assume that this translation does not effect the binary layout of the class. Thus, the encapsulation defined for the class still holds for the rest of the application—it is only for the wrapper that we allow the compiler to give us full access. As shown in Figure 4, we can inherit from A' and redefine both methods to have wrapped versions.

Although we have related the two methods (wrapped and wrapper) through inheritance, we can not use the relationship to cause an automatic invocation of the wrapper method, since the application is not necessarily available for modification (it may only exist in binary form). Rather, at runtime we use the DDL capabilities to redirect the calls to the method of the original class to the method of the wrapper class.

The code for the wrapper for $foo()$ looks like:

```
returnType wrap_A::foo()
{
    returnType result;
    . // ops before actual call
    result = A::foo();    // actual call
    . // ops after  actual call
    return result;
}
```

With this approach we do not need any special handling for the object reference itself. It is important to note that no object of the wrapper class is ever instantiated. The application creates objects of type A, calls methods on type A, and is compiled as such. At runtime, the calls to the methods which we want to wrap are redirected to the methods of the wrapper class. Although our wrapper is a subtype of A' (which we consider equivalent to A for our purposes), it does not add anything "new" in its class definition and the explicit superclass call correctly succeeds because the *this* pointer refers to an object of type A anyways.

6.2 Wrapping constructors and destructors

Constructors and destructors are special case methods that require special handling by our wrapping mechanisms.

We can not simply redirect a constructor call to the constructor of our wrapper class because when the constructor of the wrapper class is called the constructor of the parent class is called implicitly, which is again redirected to our wrapper method. This ends up in an infinite loop. To resolve this problem we write a new (non-constructor) method in our wrapper class and redirect the call to the constructor to this method. Inside this method, we would like to make an explicit constructor call, but C++ language semantics demand that an explicit constructor call should create a temporary object of the class, which is not what we need. Our solution to this issue is solved by reverting to using the *extern "C"* capability and calling the mangled symbol of the parent class constructor, passing the *this* pointer as an explicit parameter. Although not elegant, it does work.

With destructors we face a similar problem. If we redirect the destructor call to the destructor of our wrapper class, the destructor of the parent class is implicitly called after returning from the destructor of the wrapper class. Another restriction that we have in the case of destructors is that some compilers mangle the symbol of a destructor in such a way that it cannot be explicitly called, even using the *extern "C"* mechanism.

Thus, similarly to constructors, we redirect the destructor call to a regular method of the wrapper class; however, we do not need to call the destructor through a mangled C function call. Rather, C++ allows us to call the destructor of our own class (the wrapper class), and when we call the destructor of the wrapper class the destructor of parent class is called implicitly. The destructor of the wrapper class is an empty method, since no object of the wrapper class exists anyways; it only serves to cause an invocation of the parent class' destructor.

6.3 Wrapping virtual methods

The process of linking in the case of virtual methods is quite a bit different from non-virtual methods. Every object of a class with virtual methods has a pointer to the class *vtable*. The entries in the vtable point to the actual methods for the class, no matter where they are in the class hierarchy. To wrap virtual methods we need to intercept the binding of the entries stored in the virtual method table.

Generally, C++ compilers optimize away some of the vtable use on calls of virtual functions. For example, if an object variable is declared rather than an object pointer, any calls to virtual methods made on that variable (using the $o.foo()$ syntax rather than the $p \rightarrow foo()$ syntax as on a pointer call) can be resolved at compile time, since the object type is known and is static. These calls will to the linker look like non-virtual method calls, and will use normal symbol resolution and can easily be handled by DDL as previously explained.

For virtual method calls on object pointers that might point to a variety of object types, however, the vtable must be used. When supporting calls through a vtable, the mangled method names are not even referenced, and will not appear as externally required symbols for the linking process to resolve.[5]

The object vtable pointer is initialized by the constructor(s), and these are (generally) in the same object code as the vtable itself, which is (generally) a static list of symbols that are resolved at load-time, not run-time.[6] The effect of this is that the basic mechanisms of DDL run-time linking interception are completely bypassed, and thus without some new capability, virtual method calls can not in general be supported.

We have solved this problem by building into DDL a hook for watching load-time symbol resolutions. At this point we do not allow manipulation of this step, because we have not yet considered all of its ramifications, but we can record the information about symbols other than dynamically linked functions. Since these are resolved early on, by the time our initialization code is called, the vtable symbols are known, and we can over-write the vtable entries with pointers to our wrapper functions.

6.4 Wrapping and inheritance

So far we have concentrated on wrappers for individual methods of C++ classes. When we aim to wrap a compete C++ class we face some more issues. In the object oriented paradigm, a class may inherit methods from one or more base classes. So if we want to wrap a derived class we need to wrap the methods of the base class(es) too. All the calls to these methods will be redirected to the methods of our wrapper class. But in this, all the calls on methods for objects from other subclasses, which also inherit the methods and which we do not want to wrap, will also be redirected to our wrapper class.

Figure 5 shows our wrapper class in a larger class hierarchy. For example, suppose we have a class B which has a method $B :: foo()$, and two subclasses are inherited from it, $S1$ and $S2$. $S1$ and $S2$ will use the method $foo()$ of base class unless it is overridden in them. Here if we want to wrap class $S1$, we will inherit our wrapper class from $S1$ and will override the method $foo()$ in it, as shown in the figure.

The problem is that all the calls to $B :: foo()$ will be redirected to $wrap\_S1 :: foo()$, even if they are made on an object of type $S2$. This is because we are using symbol redirection on the $B :: foo$ symbol.

[5] And since it cannot be known at compile time which class' method will be invoked, it is impossible for the compiler to "pick" a symbol refer to.

[6] If a class, or classes related through inheritance, are in different shared libraries, there is some run-time symbol resolution, but this special case does not help us solve the general case.

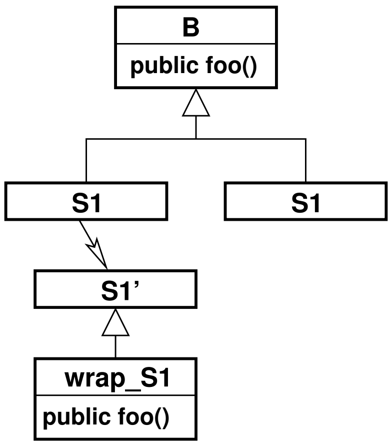


Figure 5: C++ inheritance problem.

To resolve this problem we make use of the Run Time Type Information (RTTI) provided by C++. With this we can find the type of object calling this method at runtime and bypass any wrapper processing of a call that is not made from an object of our target class. The wrapper will still add overhead because it is invoked and needs to pass the invocation on to the real method, but the functionality of the wrapper can be applied to only one subclass, even in the presence of "shared" base class methods.

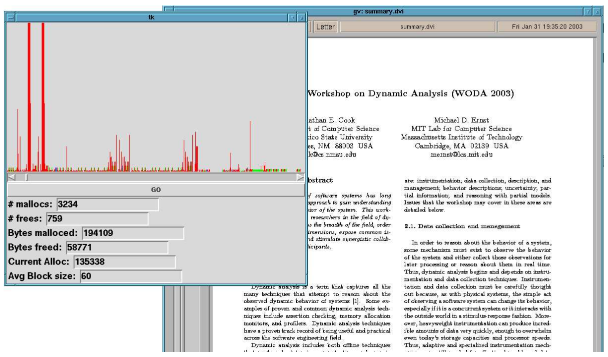
7 Example Uses of DDL

7.1 Scripted Analysis Tools

The first example tool we have built that uses our DDL framework is Datcl, a tool that allows some dynamic analyses to be written at the scripting level. This tool allows new dynamic analysis ideas to be prototyped in a high level scripting languages (Tcl), and enables even project-specific analyses to be developed cost-effectively.

In Datcl, we employ only the link interception capability of DDL. The user of Datcl writes a simple specification of the introspection points (function calls) that they are interested in observing, and a wrapper generator generates the C code that implements the interception of the introspection points, the hooks into the Tcl language level, and the call to the actual application function.

The Datclwrappers call a Tcl procedure before the actual function, and then call a Tcl procedure after the function is done. Thus, each introspection site is matched by a pair of Tcl procedures that the tool builder writes. The Tcl procedures receive the argument values used in the call, and the post-call



```
# mallocs: 3234
# frees: 759
Bytes malloced: 194109
Bytes freed: 58771
Current Alloc: 135338
Avg Block size: 60
```

Figure 6: Memory usage analysis in the shared library framework.

Tcl procedure also receives the return values.[7]

Since Tcl comes with the GUI toolkit Tk, the capabilities of Datclinclude runtime visualization of application behavior. Figure 6 shows a graphical view of memory allocation in the *Ghostview* Postscript viewer application, fully implemented in Tcl/Tk on top of Datcl, by performing introspection on the memory allocation system calls.

7.2 Component Views

In this section we demonstrate the data gathering capability of DDL, without demonstrating its link interception capability. The scripted analysis example of watching memory allocation in *Ghostview* already showed the link interception capability.

In the example here, we show the interconnections between shared objects (shared libraries, the main program, and other loaded shared objects) as established and used at runtime. This is different than what would be determined statically, because it reflects the symbols that were actually bound at runtime. With lazy first-call binding, this only shows functions that were actually used. However, as we will see there are many other symbols that are bound at load-time, and for these we cannot know in this simple analysis whether they were used or not.

Figure 7 shows the connectivity between the shared objects that make up *Adobe Acrobat Reader (acroread)*. The first (or only) number on the link is the number of first-call bindings that occurred from the source to the destination object. The second number is the number of load-time bindings that occurred. These are either shared data symbols that

[7]Datclcurrently understands only a limited number of simple data types, including numeric types, opaque pointer values, and a special string type that will convert character pointers to Tcl strings.

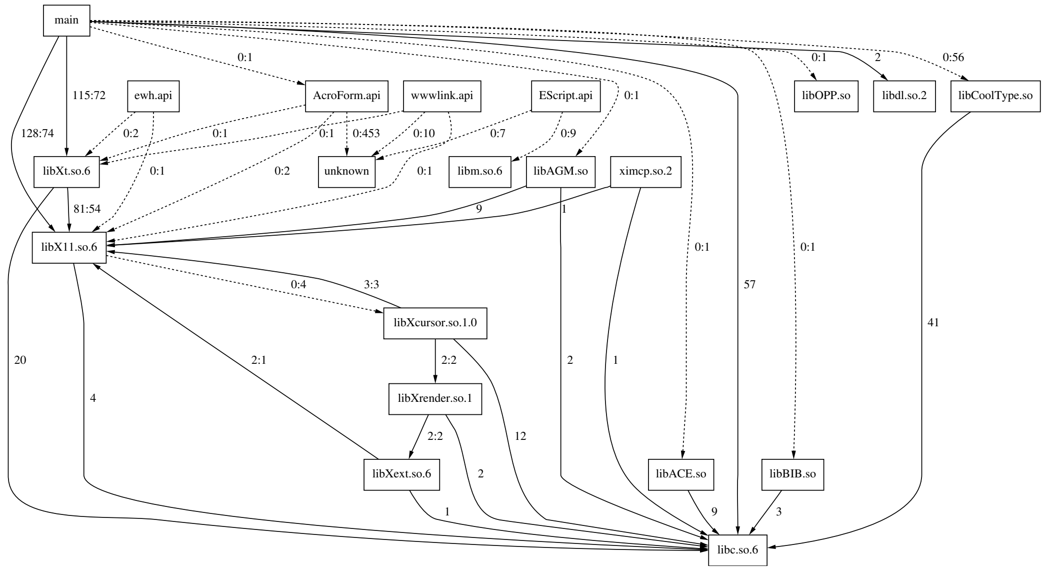


Figure 7: Acrobat Reader library interconnections (edge numbers are run-time:load-time symbol binding counts).

need bound immediately, function symbols bound by the application using *dlsym()*, or function symbols that were requested to be bound at load time. If the number of load-time bindings is greater than 75% of the total bindings, the link is drawn as a dotted line.

This application is an X-based application and so shows its centralization around the use of the X libraries. *Acroread*, however, also loads plugins and additional shared libraries through the programmatic *dlopen/dlsym* API. A number of shared objects are only connected with dotted lines (and have 0 runtime bindings) These include the *CoolType*, *ACE*, and *AGM* libraries; and plugins, identified by a name not beginning with "lib". The static library dependencies of *acroread* do not include the shared objects connected by dotted lines[8], and the main application shows a dependency to *libdl*, which contains the dynamic linker API, and so this tells us that the objects with dotted lines are being loaded by the application.

For the plugins and libraries that *Adobe Acrobat Reader* loads programmatically, it forces them to have their externally required symbols resolved immediately rather than lazily. These are the shared objects that are only connected through dotted lines. This decision makes sense when one considers application robustness, because it would be better to have a plugin (embodying some functionality extension) fail at load time rather than later at run-time, which would cause the whole application to fail.

An interesting pseudo-shared object that shows up is the "unknown" node. This is not a real object, but simply catches symbol lookups by the dynamic linker where the name of the shared object it is found in is not known. Inspecting the individual symbols, it is clear that they do belong with other libraries already known (most of the symbols are X library symbols). Yet for some reason in certain contexts the dynamic linker cannot resolve the library name. We are still working on solving this, but it should be noted that the debug output that is available on the standard linker also does not print the library name in these cases. For example, most of the large number of links from *AcroForm.api* are X windowing library functions.

These views of the interconnections can help in understanding the dependencies between the shared objects and could lead to refactoring the shared libraries into a more cohesive packages.

7.3 Example: runtime behavior modification

In this example the task is the understanding and modification of the SimpleScalar CPU/architectural simulator [3]. We were able to use DDL to understand the behavior of SimpleScalar at a function-call level, and then used DDL to modify the run-time behavior (without modifying the functional source code) to implement the switching between detailed

[8]Static dependencies are printed out using the *ldd* command.

and functional simulation modes (an important extension to the community [4]).

Using DDL's tracing capability, we found call sites in the existing program that could be to hook functional simulation into. We created an additional shared library with functional simulation code, and then used DDL to dynamically modify which functions are called at these existing call sites. We simply took the calls that entered into the detailed simulation, and dynamically redirected all but one of them to an empty-bodied function, and then redirected one (we chose `ruu_fetch()`) to a function that performs one functional instruction simulation step. When we needed to switch back to detailed simulation we simply restored the original binding of those function links. In this manner, the source code for `sim_main()` is not changed but the behavior is switched back and forth between detailed and functional simulation.

We thus accomplished a major change in the functionality of a fairly complex program without changing its functional code. Without any explicit calls to our new code, we were able to use DDL to modify the system to exhibit new, valuable behavior.

8 Reflections, Lessons, and Ideas

When first starting this project, we perhaps naively assumed that we could accomplish it in a fairly clean and portable manner. For the most part, indeed, we have done so. The changes made to the underlying Gnu source code are small, and most of our capabilities reside in external, independent libraries.

Yet, we found that distributing our DDL linker to be quite troublesome, even simply on one platform, Gnu/Linux. Most users of Gnu/Linux install binary distributions, and even most who might actually install source typically only install source for the Gnu/Linux kernel. Very few users actually bother with the source for the rest of Gnu/Linux, including the C library and its associated dynamic linker. This is a **good** thing, since any commodity system must be usable by people who do not desire to build it from source code.

However, we have found that the dynamic linker is intricately tied to the C library, and it has been almost impossible to distribute a binary linker separate from its binary C library that it was compiled with. There simply seems to be too much interdependency between these to actually make a binary DDL available to a wide variety of Gnu/Linux users. This statement is true even within a single version of glibc, since the various Gnu/Linux distributions might apply different patches, and inevitably have slightly different build environments.

For example, there is a symbol "*rtld_global*" that is owned by the dynamic linker but accessed by code in the C library. It is a reference for a long data structure that contains many fields of global data items that might be accessed by other code. That is not all bad—however, there are conditional compilation units in it that change what fields are defined, and thus where the rest of the fields are in relation to the start of the structure. So while it might look clean to simply export one symbol in lieu of many, in actuality it makes the interaction very brittle. We have found that it is almost always necessary to run the DDL linker with the libraries it was compiled with.[9]

We are not interested in "bashing" glibc from afar—rather, we recognize that glibc is an extremely robust and heavily relied-upon system, and all of us would wish to have software so heavily used and appreciated by the world. Yet in experimenting with it in ways that others perhaps have not done very much, we can find areas that might be worth looking into improving.

8.1 Ideas for Improvement

From an outsider's perspective, it would seem obvious at first thought that a dynamic linker ought to be considered a piece of an operating system rather than a piece of a language library. Indeed we were initially surprised to learn it was part of the glibc distribution. One might not be too surprised to find a static linker as part of a compiler distribution, but the dynamic linker as part of a library? Since it is acting as much like a loader as a linker and is dealing with run-time configuration of a process, one would suspect that it would be an operating system component.

Yet, we recognize the blurriness between these pieces of a system. Given that the API to the operating system (e.g., the POSIX API) is typically implemented as part of the C library, and that there is a POSIX interface to dynamic link functionality (*dlopen()*, etc.), the dynamic linker does need to have some interaction between itself and the C library.

Practically speaking, it would seem highly unlikely that the Gnu dynamic linker would become a component distributed separately from the C library. However, it does not seem unreasonable to aim for a goal of making it robustly independent of a particular build of its associated C library. Being

[9]When we started, with glibc 2.2.5, things weren't quite as bad and we regulary could distribute binaries. Since moving to glibc 2.3.2 and beyond, the dependencies seem worse.

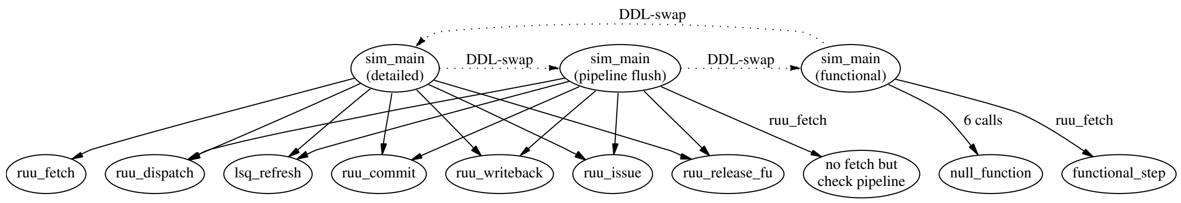


Figure 8: Implementation of *SimpleScalar* behavior modification.

able to offer binary distributions of the Gnu dynamic linker separate from binary kernel and library distributions would encourage more projects like DDL to move the shared library platform towards a more configurable, dynamic, manageable execution platform.

Our experience in building DDL has not yet reached a level where we *know* all the interactions between the dynamic linker and the C library, but we hope to investigate this interaction in detail and offer suggested modifications to make the dynamic linker a true independently deployable component.

9 Related Work

The DITools project [13] is the closest related work to our DDL project. They used a similar approach to link interception and modification, and supported redirecting a link to a wrapper and also an event notification mechanism where each monitored call was not wrapped but did generate an event to a fixed-interface callback. It does not appear that they addressed the issues surrounding C++, nor did they do non-function symbol resolution nor runtime link modification.

Ho and Olsson [8] describe *dld*, a tool for "genuine" dynamic linking. Their tool provides the capability to load and unload shared libraries, breaking links when a library is unloaded and relinking them to new code when new libraries are loaded. However, it does not appear that they ever supported redirection of links to different symbol names.

Hicks et. al [7] work on binary software updating from a formal perspective. Their methods use typed, proof-carrying assembly code from which they can verify that an update will be safe. Their infrastructure includes special languages and compilers to generate the annotated assembly code, and a runtime framework that uses it.

Additional systems that provide instrumentation capabilities on executable binaries exist. Dyninst [2] can patch custom code into pre-existing executable code, and has provided a platform for several research tools. Valgrind [14] provides a complete simulated CPU and execution space to the program under inspection, and is extensible, thus allowing new dynamic analyses to use it as a foundation.

There is much work in dynamic introspection and modification of Java programs, but since this work is in a very different environment than ours, we do not explain it in detail here. Some representative references are [5, 6, 10, 12].

10 Conclusion

We have presented DDL, an extension to the standard dynamic linker that allows introspection and modification of the dynamic linking process. This capability supports a wide range of uses for software engineering practitioners and researchers, including a foundation for runtime monitoring and dynamic analyses, dynamic runtime program evolution, and other ideas that can use control over the linking process.

In our future work we are pursuing the use of this platform to support multi-version software fault tolerance and evolution, and to support our ongoing research interests in dynamic analysis. We are building an event-based extensible tool framework on top of DDL, called ETF, and we have an initial prototype of C++-based execution of multiple versions of classes, along with class and object evolution.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-0306457, EIA-9810732, and EIA-0220590. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] N. Abbas, S. Tambe, R. Srinivasan, and J. Cook. Using DDL to understand and modify Sim-

pleScalar. In *Proc. 2004 Working Conference on Reverse Engineering*, page to appear, Oct. 2004.

[2] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000. www.dyninst.org.

[3] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. *Technical Report 1342, Computer Sciences Department, Universi ty of Wisconsin*, June 1997.

[4] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. *4th IEEE International Workshop on Workload Characterization*, December 2001.

[5] M. Dahm. Byte Code Engineering Library. 2002. http://jakarta.apache.org/bcel/.

[6] S. Eisenbach and C. Sadler. Changing Java Programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 479–487, Nov. 2001.

[7] M. Hicks, J. Moore, and S. Nettles. Dynamic Software Updating. In *Proc. 2001 ACM Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[8] W. Ho and R. Olsson. An Approach to Genuine Dynamic Linking. *Software Practice and Experience*, 21(4):375–390, 1991.

[9] J. Levine. *Linkers & Loaders*. Morgan Kaufmann, San Diego, CA, 2000.

[10] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proc. European Conference on Object-Oriented Programming*, pages 337–361, 2000.

[11] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[12] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. In *Proc. 2002 International Conference on Software Maintenance*, pages 649–658, Oct. 2002.

[13] A. Serra, N. Navarro, and T. Cortes. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In *Proc. 2000 Usenix Technical Conference*, pages 225–238, June 2000.

[14] J. Seward. Valgrind, an Open-Source Memory Debugger for x86-Gnu/Linux. Technical report. valgrind.kde.org.