An Extensible Infrastructure for Monitoring and Managing Applications in a Shared Library Environment

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

Abstract

The existing platform of shared, dynamic link libraries has been long overlooked in its potential for providing to developers such capabilities as management, configurability, and monitoring. With the proper support, the dynamic link mechanisms can be exploited to support many CBSE and software architecture ideas, and can provide a platform for monitoring and self-management capabilities.

This paper describes ETF, an extensible tool framework for such support that is built on top of DDL, our extended, open dynamic linker. ETF offers adaptation, evolution, and autonomic management support to systems built on the dynamic link library platform.

1. Introduction

Software systems deployment has made a rapid march from almost no run-time management facilities to more and more dynamic management capabilities. Run-time frameworks have been central in this march, and now many such frameworks include the capabilities for managing and monitoring the applications running on them. The frameworks give the developers and deployers the low-level capabilities from which higher level tools and applications can be built. Perhaps best known is the Java environment, with introspection and reflection capability, customizable class loading, and now a programmable "debugger" API that gives control over the application.

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.

Yet most of these frameworks support new applications and do not look back to what legacy applications might need. It is our position that the existing deployment framework of shared, dynamically linked libraries (DLLs) has the fundamental potential of supporting the same kind of management capabilities. The DLL framework can be a true software component deployment platform, and can offer the capability of building self-management into legacy software systems.

Dynamic linking is most often seen as a way to save memory resources, both secondary (by having smaller executable files) and primary (by applications sharing code pages). Yet, its fundamental decoupling of a system into separately deployable components offers much more. Recent use of this has become popular in browser plug-ins, which are essentially shared objects that obey a strict API dictated by the plugin standard. Using components with custom interfaces within a management framework, however, is achievable on top of this basic technology.

To this end, we have been building ETF, an extensible tool framework based on DDL, a customized dynamic linker that offers programmatic access to the linking process, and enables dynamic manipulation of the existing bindings in a running software system. This paper summarizes DDL and then presents ETF in detail, and explains how it can support both monitoring and management tools, and the deployment of CBSE and system architecture composition ideas.

The following section (2) gives a brief overview of DDL, Section 3 presents the overall architecture and design of ETF. Section 4 discusses how ideas from software architecture and CBSE are supported by ETF. Section 5 presents real example uses of DDL/ETF for monitoring and manipulation of application systems. Finally, Sections 6 and 7 present related work, conclusions, and ideas for future work.

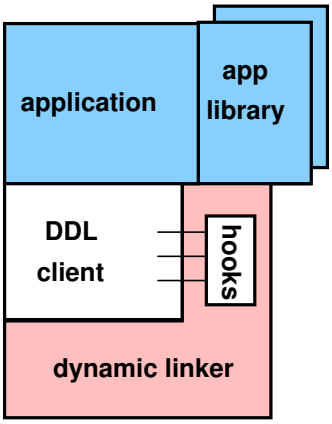

Figure 1. DDL system architecture.

2. DDL: a dynamic dynamic linker

Here we present a brief overview of DDL, a framework of modifications and extensions to the dynamic linker that allow us to have dynamic control over the linking process, and to implement a range of features desirable for component frameworks and self-managing applications. DDL allows powerful control over the linking process, 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 linker/loader, which is part of the Gnu C library. Our current tests have only been on the Gnu/Linux platform, although the Gnu libraries (and the dynamic linker) are ported to many other platforms.

Figure 1 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.

In building DDL, we wanted to be very minimal in our modifications to the existing linker functionality, and wanted to be able to have DDL revert back to the basic dynamic linker when needed. Thus, at the lowest level, we added simple hooks into the dynamic linker itself so that we could interact with the linking process, yet not change the linker code hardly at all. These hooks are implemented as callbacks into a preloaded client shared object. If no client tool is loaded, or an environment variable is left undefined, the dynamic linker skips the callbacks and simply executes its native, existing functionality.

The fundamental capabilities that DDL supports is link announcement, interception, and redirection. This allows DDL and the tools that use it to peer into the dynamic linking process, collect information about it, and control it. When a symbol is being looked up for purposes of linking, our hooks in the dynamic linker perform callbacks into the DDL client. 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 the DDL client. The DDL client is in essence passive and event-driven, those events being, for the most part, link requests.

A client tool can simply record information about a symbol and a link, but generally tools will want to do more than that. The client has the option of redirecting the request to some other symbol. This allows a client tool to, say, redirect a link to a wrapper function, which will perform some tracing or monitoring computation and then call the original function. Since DDL is dynamic, the original function can still be bound with the dynamic linker, thus avoiding much of the pain of using existing preload mechanisms to implement wrappers. DDL can also do much more than the static preload mechanism, which is documented elsewhere [17].

Link information that is announced through the DDL callbacks include the string symbol being requested, the shared object name originating the request, and the address of the jump table entry that will be modified. After a symbol is resolved, another announcement is made through DDL that contains the resolved symbol, the shared object it was found in, and its address. Note that announcing the jump table entries that correspond to link requests allows a tool to record those entries and later modify them directly if it wants. Thus DDL supports dynamic reconfiguration of an application.

3. ETF: an extensible, event-based tool framework

DDL is a tool that fundamentally opens up the linking process, but it is quite low level and primitive. Thus, further supporting infrastructure is necessary for truly enabling monitoring and management of applications.

In considering what support may be desirable, we began with a typical API approach, and thought that we might just build services that potential tools might use. However, it became clear that multiple tools might be needed at a single time. For example, if a management tool is being used to allow reconfiguration, a monitoring tool that collects data from one piece of the system should also be able to be deployed within that system.

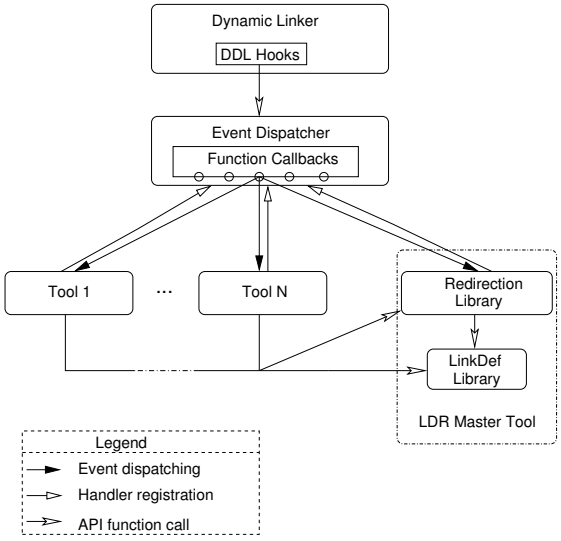

Figure 2. The ETF tool framework.

To this end, we created ETF, an event-based extensible framework for allowing multiple tools access to the DDL capabilities. This is shown in Figure 2. Tools, each embodied in a shared object, register with the event dispatcher to be notified of link request events. The event dispatcher is what interacts with the base DDL support, rather than the tools themselves.

We envision that management or monitoring tools would be implemented as one or more cooperating "mini-tools". This idea, of course, is not new, being used as far back as the Field environment [14], albeit at a different level. In ETF, if one wants to plug in a tool that watches a particular class and checks invariant properties, this tool should not interfere with other tools that might be doing other things.

The event dispatcher itself is plugged into DDL, and treats the link callbacks as events to be published to the tools. These events are load time symbol lookups and definitions, and run time symbol lookups and definitions. Because these events are central to ETF and expected to be heavily used, they are supported by specific, dedicated event handlers in the tools. In ETF, event handlers are functions in the tool shared object which the tool registers with the ETF event dispatcher.

When a shared object is loaded, it generally has a set of external dependencies (symbols) for which it needs load-time binding. These include global variables, internal C++ symbols such as vtable symbols, and others that are not "called" and cannot support lazy binding. These non-code symbols are announced through DDL/ETF at symbol lookup time and when the definition is known. This typically happens before the application begins execution (i.e., before *main* is invoked), but it can be later if a shared object is *dlopen'ed* by the application. While it is possible to redirect a non-code symbol, we have not yet used this in practice. Monitoring and recording the bindings are useful, however, for data monitoring and for access to C++ vtables, another jump table mechanism that we consider for manipulation.

Alternatively an object, or the environment, can request that all symbols be resolved at load time. This might be done for a time-sensitive system to avoid the first-call overhead of lazy binding, or for an assurance that all needed bindings will be found before execution begins.

Later run-time bindings, which are the general code symbol binding mechanism that shared objects use, are the main target of our tools. Here is where, at symbol lookup time, a tool may be expected to substitute a different symbol in order to redirect the binding. When the binding is being finished after a symbol definition is found, an announcement event is sent to the tools with the full information.

This separation of lookup and definition events is necessary for binding redirection by tools. Tools need to see what symbol is going to be looked up so that they can substitute an alternative in its place. After a symbol is resolved, another event is needed to announce the definition of the symbol. Note that if a tool has substituted a different symbol, the announced symbol being looked up will be different than the announced symbol definition.

Symbol binding redirection is the main mechanism supported by DDL/ETF for a tool to insert a monitoring or manipulation point into the application.

3.1. Centralized link/definition/redirection management

Along with the link interception, we maintain an internal data structure of resolved symbols (definitions), and the bindings or links that refer to them. Maintaining this information during the runtime of the program allows us to support dynamic program evolution through runtime link modification (redirection). This is done in the LDR master tool, a client tool of ETF that is made available to other tools through a direct API, since this capability is generally useful, should not be re-built in every tool, and does not need to be event-based.[1]

1 We may, however, replace this API with an event-based interaction in the future simply for orthogonality, if it can be done clean and efficiently.

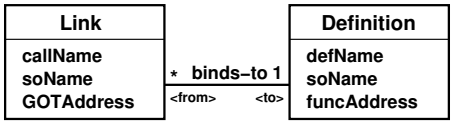

Figure 3. Link and definition UML.

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

In order to modify a link, we simply need to change the address that is in its jump table 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 jump tables are allocated per shared object (the main program and other shared libraries), and so 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.

Because the events described so far are triggered by the dynamic linker, apart from startup binding and lazy link resolution and binding, the dynamic linker and thus our tools are never going to be invoked. Therefore, we have to have some way of regaining control over the execution of the program in order to perform run-time link modification. We currently employ the OS's signal mechanism to accomplish this, assuming we can co-opt an unused signal. Some external source, be it a person or an automatic management system, would use the signal as a means of activating the ETF framework, and the tools using it. Our installed signal handler generates an event, and the tools that wish to take an action such as link rebinding can then do so. Once the application resumes execution, these new bindings will have immediate effect (when they are used).

Although we have much future work to make this type of program evolution generally useful, such as concerns about state corruption or migration, interference between existing calls to the old bindings and new calls to the new bindings, recursion, and the like, ETF offers the foundation for bringing dynamicity and run-time configurability to the legacy framework of shared libraries.

3.2. Tool events

Tools themselves can also generate events and subscribe to each other's events. This allows the constructive ability to forge new capabilities without rewriting tool capabilities. The main issues in supporting tool events are descriptive power and efficiency of delivery. While link requests and reconfiguration events and the like are not occurring too often, if a tool decides to publish events for every call to some often-used function, or for every read of some variable, efficiency does become a concern.

Because tools all share the same process space, our current design is to have a read-only event description structure whose reference is passed to the event handler, along with a reference to the event data. In this way the event data does not need parsed every time, only traversed to extract the information desired by any given tool. Tool event types are designated by unique integers.

Tools can choose to register a centralized event handler for these events, or can choose to have specific handlers. For tools that are intended from the start to cooperate, they will already know each other's event formats, and can register unique handlers for these events.

3.3. Tool conflicts

One issue with allowing multiple tools to operate is handling conflicts in their requests on application monitoring or management.

In monitoring, two tools might wish to wrap the same function. If the wrappers are read-only, they can be stacked and will not interfere, other than with more overhead. If tools install a wrapper that might modify parameters or return values, this would possibly conflict with another tool's needed monitoring or management. In run-time modification, two tools might ask to modify the same link in different ways; or one tool might request to modify a link that another tool has already wrapped for monitoring.

Our solution to handling these conflicts is to require tools to declare a priority for themselves, and to require tools to declare whether a link redirection will be a read-only wrapper that will support the original functionality. If tools are only installing read-only wrappers, we allow these to be stacked. Otherwise, higher priority tools will supersede requests by lower priority tools.

3.4. Tool threading and external tools

The dynamic linking foundation, and our framework on top of it, necessarily sits within application threads. In regular dynamic linking, the thread initiating the as-yet-unresolved call implicitly invokes the dynamic linking functionality, which resolves the needed symbol and then jumps to the actual function. When we build on top of this, extending the linking functionality, adding wrappers or intermediaries between re-

quired and provided services, we insert more overhead directly into the application threads of control.

While there is no way to completely avoid overhead within the application threads, ETF supports the creation of in-thread tool proxies, which then interact with separately threaded tools. In this manner, we can add significant tool functionality without "unneeded" application slowdown. Obviously, some monitoring and management tools will need to control the execution of the application, and even force synchronous interaction (such as stopping an invocation until security checks on parameters are done), but other tools, such as visualization tools, that have external needs asynchronous to the application computation can indeed be constructed on top of ETF.

Tool proxies can also be used to communicate with tools external to the application process. Whether the tool is too heavyweight to combine with the application or there are existing tools that can take a data feed from another system, we can create proxies on top of ETF that respond to events by passing them on through IPC or other communication facilities. One possible use of such a capability is the remote monitoring and management of a system not originally built to support such.

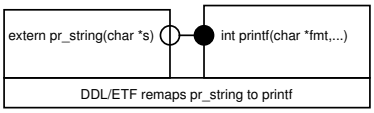

Figure 4. Direct null connector by remapping symbols.

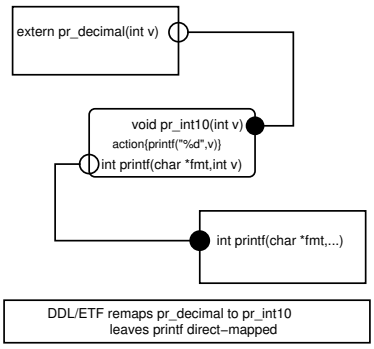

Figure 5. Complex connector by interposing a connector component.

4. Deploying architecture ideas with ETF

The area of software architecture codified the ideas of specifying and reasoning about the large-scale structure of a system. Central in this work are the notions of components and connectors. While much of the architecture works delves deeper than these two simple ideas, the notion of connecting up components is core [11].

In traditional programming languages, these ideas are still foreign. Components in programming languages, be they functions or classes or packages, are not able to refer to their external dependencies using their own internal namespace. Connections are bound by global name agreement, because the undefined symbol from one object is resolved by finding the exact same symbol in another object. This makes controlling the global namespace very important.

Dynamic linking, being an offshoot of traditional program linking, has taken the same view of a system that programming languages have. Our ETF tool breaks that barrier and opens up the linking process to allow new mechanisms for system composition.

With ETF, one can view a shared object's external symbol dependencies as locally named port declarations. The dynamic linker's job, then, is to bind these ports with other locally named ports on other shared objects, using some type of connector.

The simplest connector is a one-to-one null connector, which would be accomplished by simply replacing the undefined symbol being looked up with a different one, namely the symbol from the shared object providing the service. This is shown in Figure 4. Thus, the undefined symbol becomes a required port name, and the exported symbol becomes a provided port name, and ETF enables the mapping between them. Of course, both ports need to have the exact same call and invocation format in order for this mapping to work.

True architecture support necessitates the capability of supporting complex connectors. Such connectors can have computational capabilities, whether to transform some data, enforce contracts, or handle incompatibilities between endpoint interfaces. A complex connector can almost be viewed as a component itself, but the difference is generally that the connector does not embody application logic.

In ETF, complex connectors must be embodied in some functional code, and have their own provided ports and required ports. To insert a complex connector using dynamic linking and ETF, the symbol remapping is done in such a way that the connector is placed between the required and provided ports of the application components. This is shown in Figure 5.

4.1. Dynamic reconfiguration

Although the mechanisms supporting dynamic linking have been used up to now in a static fashion—that is, linking is done once and is stable for the rest of the execution—it does not need to be limited to this.

The basic support that shared objects give the dynamic linker is a *jump table* that is filled in by the dynamic linker with addresses of resolved symbols. This jump table serves as a centralization point for external references, and thus it can also serve to allow for dynamic reconfiguration. Each shared object has its own jump table, and thus is independently (re-)configurable from the other shared objects.

In ETF, we maintain information on all link sites (entries in jump tables) and all symbol definitions, and the current associations between the two. This allows us to dynamically change any link at any time during the system's execution. This moves the dynamic linker framework one step closer to being a dynamic component management framework.

Using this capability, we can insert and remove monitoring capability, we can load a new version of a shared object and update the references, and can support other forms of dynamic systems management, hopefully even some of the ideas coming from the autonomic computing area.

4.2. Limitations

Using shared objects and dynamic linking does have its limitations in terms of fully deploying component-based system ideas.

For one, dynamic linking occurs within a single process. As such, shared objects are not truly protected from each other, and many well-known accidental or malicious anomalies can occur. Shared objects have implicit access to each other's address space, and although the linker can arbitrate between symbol-based access requests, it cannot prevent or monitor implicit or accidental access.

The invocations between shared objects, since they are essentially just function calls, are synchronous. While our framework can build on top of this to implement asynchronous interaction, this would have to be designed into the shared objects that used it. Legacy shared objects, expecting synchronous interaction, would probably not support such large interaction changes.

Dynamic linking, like static linking, is done without regards to type signatures. It is assumed that the compiler has checked this already, and so when resolving symbols, only the symbol name matters. By opening up the dynamic linking process and allowing diverse binding possibilities, interface checking (at least type checking but also perhaps semantic checking) becomes important.

Currently we have not yet tackled this issue. However, there are several possibilities. Most executable formats are extensible; debuggers take advantage of this to include much information about a program, including type information. We could take a similar approach to include only the type information about the external interface of the shared object. An alternative would be to require a separate specification of the interface type signatures, similar to a header file. Methods such as how C++ embeds type information into symbol names could also be used, although this would need augmented with type relations.

Finally, not all shared object interactions need to be mediated by the dynamic linker. Implicit interactions might occur. Addresses, including functions pointers, can be passed between shared objects and then used to directly access some "unknown" exported behavior or data. Some limitations on what a shared object could do to be considered a manageable component would be required, and most of these could probably be checked with some static analysis techniques.

5. Example uses of DDL/ETF

In this section we present example uses of DDL/ETF in analyzing, monitoring, and modifying applications.

5.1. Example: Tcl scripting plugin

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 DATCL wrappers 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 argu-

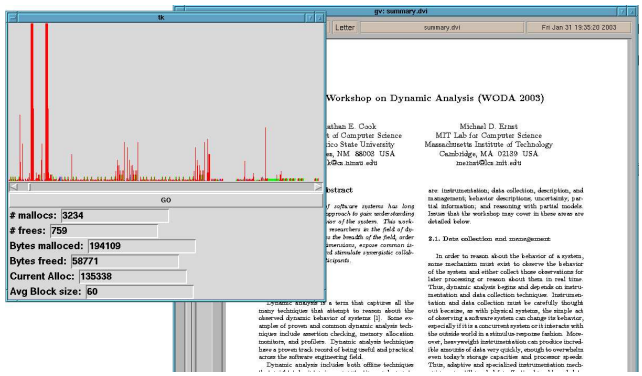

Figure 6. Monitoring memory usage in Ghostview.

ment values used in the call, and the post-call Tcl procedure also receives the return values.[2]

Since Tcl comes with the GUI toolkit Tk, the capabilities of DATCL include 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.

5.2. Example: application tracer plugin

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 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.

The application was lightly exercised before the definitions and bindings were dumped (using the OS signal mechanism in ETF). Thus, this picture captures a snapshot of the runtime architecture of the shared object components. *Acroread* 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[3], 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,

2 DATCL currently 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.

3 Static dependencies are printed out using the *ldd* command.

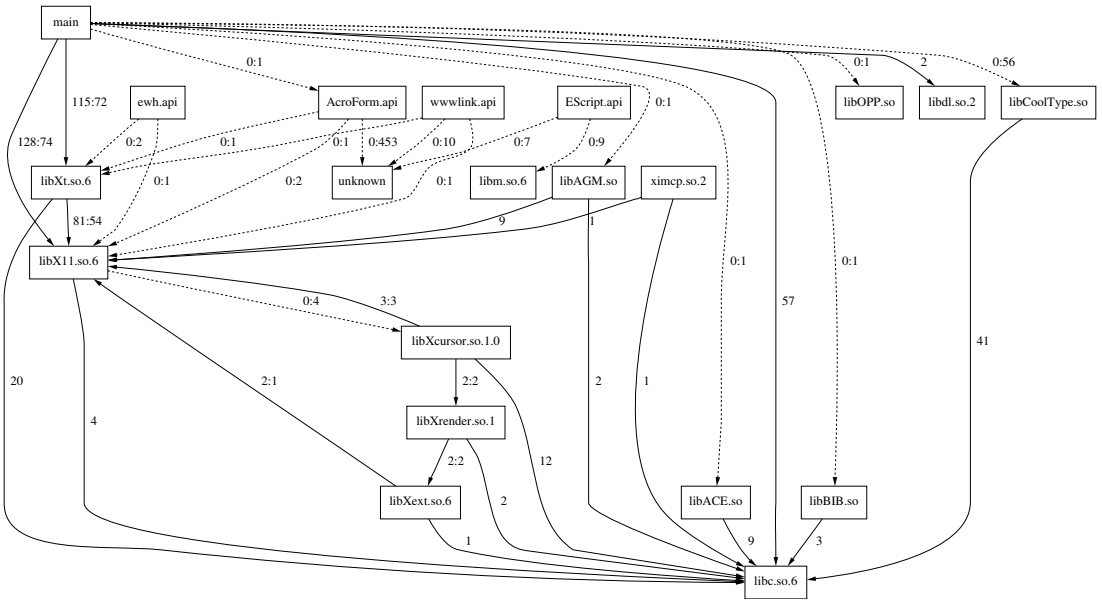

Figure 7. Shared library connections in Acrobat.

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.

5.3. 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/ETF to understand the behavior of SimpleScalar at a function-call level, and then used DDL/ETF to modify the runtime behavior (without modifying the functional source code) to implement the switching between detailed and functional simulation modes (an important extension to the community [5]).

We used DDL/ETF to generate the dynamic call graphs of both `sim-safe` and `sim-outorder`. DDL/ETF allows us to do this easily by using table-based redirection. In this mode, DDL/ETF can allow the tool builder to use a single tracing wrapper around all traced function calls. Since these two simulators are related, we then wanted to see the differences in their call graphs, in order to understand them a little better.

Figure 8 is a view of the parts of `sim-outorder` that are not in `sim-safe`. We see that in `sim-outorder`, `ruu_dispatch()` calls `mem_access()` exactly the same number of committed loads and stores that the simulator reports , and the number of memory pages used is that same as the number of calls to `mem_newpage()`. The `sim-outorder` simulation calls its main sub-functions each 52,382 times, which is one more than the number of cycles simulated because of the way the simulator finishes the program.

With DDL/ETF we can dynamically modify which functions are called at these call sites. We simply take the calls that enter into the detailed simulation, redirect all but one of them to an empty-bodied function, and then redirect one (we chose `ruu_fetch()`) to a function that performs one functional instruction simulation step. 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.

Our initial attempts at naively switching were not successful. Essentially, before switching from detailed to functional simulation, we needed to finish simulating all the instructions that were already in the CPU's pipeline. Switching now needed three modes rather than just two: detailed simulation, flushing the pipeline, and functional simulation. Since functional simulation did not do any hardware simulation and thus always finished the instructions it "fetched", there was no patch-up mode needed in going from it back to detailed simulation. We thus accomplished a major change in the functionality of a fairly complex program without changing its functional code. Without any ex-

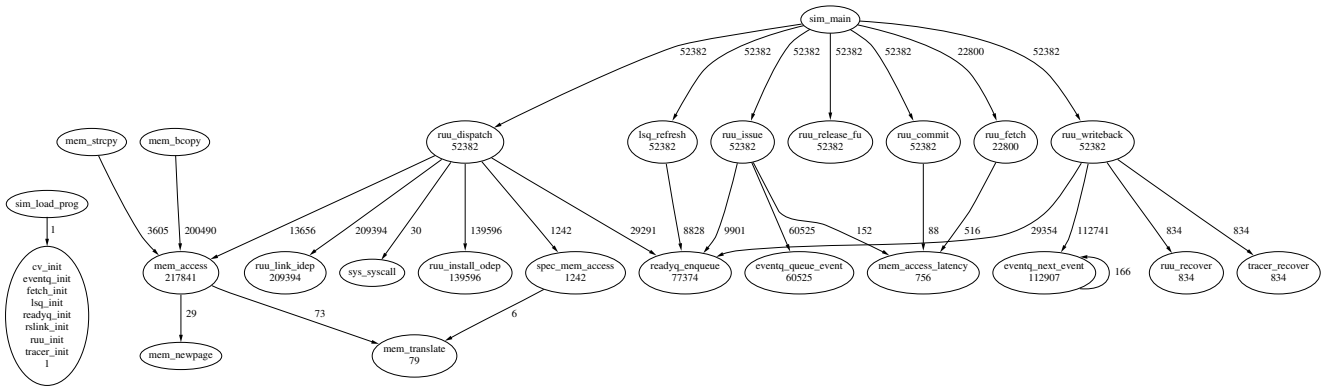

Figure 8. Differences between simple and detail SimpleScalar.

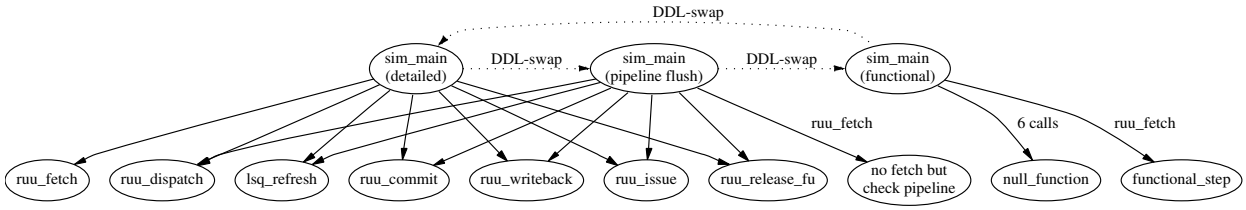

Figure 9. Implementation of SimpleScalar behavior modification.

plicit calls to our new code, we were able to use the DDL/ETF framework to modify the system to exhibit new, valuable behavior.

6. Related work

The DITools project [15] 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 [9] 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 [8] 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 [16] 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 [6, 7, 10, 13].

In the commercial world, .NET seems to offer extended capabilities beyond simple dynamic linking [12]. .NET "solved" the DLL incompatibility problems by requiring shared objects to reference exact versions of other shared objects, and even allowing multiple shared objects in an application to use different versions of some other shared objects. External rules can allow a different version to be declared compatible, so that application upgrading is possible, but tightly controlled. .NET also has strong introspection capability, and a de-

bugging API that gives control over an application, so it is possible that they support a large amount of what DDL/ETF does. However, we have not seen any indication that .NET would allow programmatic symbol redirection (although the debugger API does support run-time CLR binary editing), and there are some indications that .NET still suffers from some versioning problems [1].

7. Conclusion

This paper has describe an extensible event-based mechanism for manipulation and monitoring of an application built on shared libraries through interacting with the dynamic linker.

Shared, dynamically linked libraries have been around for quite some time, and yet they have been ignored as a platform for CBSE ideas. We believe that this ubiquitous platform can support much more dynamicity and component management than it currently does, and we are working to achieve these goals. Our ultimate hope is that we can influence the direction of future dynamic library infrastructure to include the support needed to make shared libraries true manageable components.

As with any opening of an application framework, security does become a concern. However, since some dynamic library platforms already allow redirection through the preload mechanism, we see our work as *enabling* further security measures rather than opening new holes. In future work we would like to implement an authentication mechanism to ensure that shared objects are from a trusted source. In this way we can allow management and manipulation and at the same time have confidence that the manipulation is not being done by a malicious tool.

Our current focus is in the deployment of the HERCULES framework on top of DDL/ETF, but we are also using DDL/ETF for dynamic analysis work (especially scripting language support), for dynamic behavior adaptation, and other applications. HERCULES is a framework for reliable evolution of a system where multiple versions of components can be active in a system at any given time [4]. We have an early prototype of this already working, where components are C++ classes.

Presently, DDL is stable and ETF is in a prototype stage, with thread and tool priority support being completed. Both are freely available for research use at http://www.cs.nmsu.edu/please/ddl/index.php.

References

[1] A. Berglas. Warning: .NET Hell and Version Control, unstable, irreproducable bugs. 2004. http://www.codeproject.com/Purgatory/DotNetHell2.asp.

[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 and J. Dage. Highly Reliable Upgrading of Components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 203–212, May 1999.

[5] 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.

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

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

[8] 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.

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

[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] Microsoft .NET Framework Developer Center. 2004. http://msdn.microsoft.com/netframework/default.aspx.

[13] 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.

[14] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.

[15] 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.

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

[17] S. Tambe, N. Vedagiri, and N. A. J. Cook. DDL: Customizing the Dynamic Linking Process for Program Analysis and Evolution. Technical report, New Mexico State University, 2004.