Infrastructure for Making Legacy Systems Self-Managed

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

Software systems that are successfully deployed and used seem to always have a longer lifetime than was originally expected. It is also common knowledge that the cost of maintaining and evolving those systems during that lifetime dwarf the initial cost of creating the system. This makes support for self-management in the legacy software arena all that much more important.

We are building an infrastructure for such support, called DDL, that offers adaptation, evolution, and autonomic management support to systems built on the dynamic link library platform. With the proper support, dynamic link mechanisms can be exploited to support many CBSE and software architecture ideas, and can provide a platform for self-management capabilities. DDL is meant to provide that support.

1. INTRODUCTION

Self-managing systems need infrastructure to support introspection and manipulation of themselves. And while the "start from scratch" approach is attractive, allowing one to define their own customized framework (and perhaps even a new programming language) and not to worry about legacy issues, there is a tremendous amount of existing software that could benefit from self-management. The question is: can self-management be brought to legacy platforms?

It is our position that the existing deployment framework of shared, dynamically linked libraries has the fundamental potential of being 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 libraries (also called shared objects) that obey a strict API dictated by the plugin standard.

We believe that a true component framework can be realized using shared objects, if the underlying foundation would support it. To this end, we have been building such framework capabilities based on DDL, a customized dynamic loader, that offers programmatic access to the linking process, and enables dynamic manipulation of the existing bindings in a running software system. Such capabilities will be necessary if self-management ideas are to be deployed into these legacy systems. This short paper summarizes the capabilities of DDL, focusing on how those features can support self-managing systems.

The following section (2) gives a brief overview of DDL, Section 3 discusses how DDL can support both the diagnosis and repair/reconfiguration aspects of self management. Finally, Section 4 presents related work and Section 5 concludes.

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.

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 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.

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 implemented an application services level that provides even higher level interaction for some

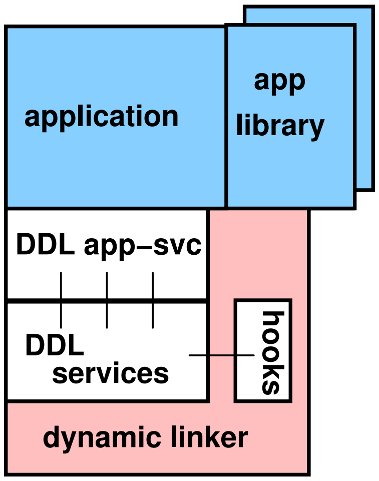

Figure 1: DDL **system architecture.**

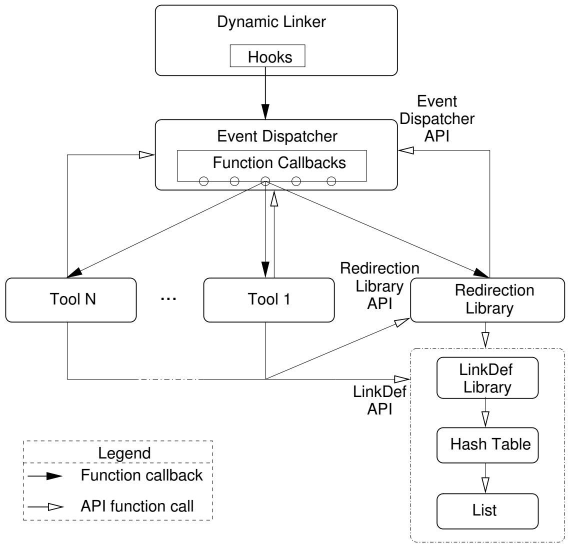

Figure 2: DDL **event-based tool framework.**

types of common services—one such service is scripting language support.

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, 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 control library. 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 generally passive and event-driven, those events being, for the most part, link requests.

We have created 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. We are currently extending this to allow the tools themselves to interact with each other through the same event mechanism. We envision that self-management frameworks would be implemented as one or more cooperating tools.

With the link interception, we maintain an internal data structure of resolved symbols (functions and global symbols), 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. This is done in the redirection library that acts as a master control tool and is made available to other tools through a direct API, since this capability is generally useful and should not be re-built in every tool.

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.

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 runtime link modification. We currently employ the OS's signal mechanism to accomplish this. Our installed signal handler reads a remapping specification, and rebinds the links as directed. Once the application resumes execution, these new bindings will have immediate effect (when they are used). In later sections, we will see other ways that self-managing applications can regain control.

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, DDL offers the foundation for bringing dynamicity and runtime configurability to the legacy framework of shared libraries.

3. SELF MANAGEMENT USING DDL

Figure 3 represents the basic abstract structure of a self-managing system. A feedback loop is essential for an autonomous system. Some decision/action component must be able to examine data from the executing system, diagnose problems and decide on actions to modify, re-configure, and/or update the system, and then be able to perform those modifications. DDL supports both the data collection ability for diagnostic purposes and the system modification ability for adaptation purposes.

3.1 Supporting Self-Diagnosis

Self managed systems must be able to inspect their operation and diagnose situations that must be addressed. Two basic approaches to this issue is to have either in-line, in-process diagnostic checks or an observer process that concurrently inspects the system and updates diagnostic computations appropriately. DDL supports both types of operations, although the former is more directly supported.

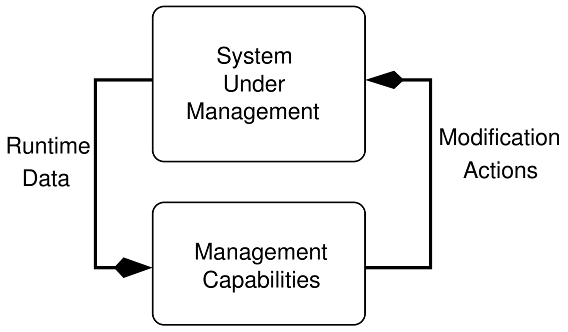

Figure 3: Self-managing system.

3.1.1 *In-line Diagnosis*

Since DDL is mainly concerned with the bindings between shared objects, it naturally supports inserting wrappers in between the invocation and the real target of the invocation. In this way, pre- and post-invocation diagnostic computations can be placed in the wrapper. While dynamic linking generally assumes a 1-to-1 mapping between invocations and targets, DDL supports a table-based rebinding mode that enables the re-use of a diagnostic computation for multiple targets, so that things such as class invariant checking are possible without a unique wrapper for each method of the class.

DDL allows monitoring of computation events—essentially invocations of public code units. However, it can also make available any global external symbols that are visible to the dynamic linker. Thus, some global state is accessible. In a C++ setting, object data is also accessible, since invocations carry the object reference with them. DDL is certainly not the complete introspection solution, but it does offer easy implementation of some system probe points.

With in-line diagnostic checks, one must take great care in what is being evaluated where. In complex situations, such as monitoring data structure integrity, it is easy to insert multiplicative terms into the overall complexity of the underlying application algorithms. In an autonomous self-managed system, the diagnostic techniques might involve very complex "AI" decision algorithms, and in-lining these within the application computation would mean that efficient incremental checking algorithms are needed. We are not working in that area, and leave it up to the engineer to know how much diagnostic computation can be done in each specific wrapper. This does lead, however, to the desire to place the management into a separate thread or process.

3.1.2 *Observer Process Diagnosis*

If diagnostics are to be computed in a separate observer process (or thread), DDL can support this in an event-based manner. In this scenario, DDL would insert wrappers that simply convey an IPC message indicating an event that takes place in the system. This IPC messaging should allow an acknowledgment to be returned before the system proceeds with its computation. In this way the observer process can choose to provide a default immediate ack and then process the event concurrent with the system (and taking a chance that the system proceeds in error), or to delay the ack until it decides that no action needs to be taken.

Since the observer portion of a self-managed system should ideally have full access to the application portion, it would be natural to think of the observer as a thread rather than a heavyweight process. The observer thread, however, should not be considered as symmetric to the application thread(s), since it should have full access to the application (code+data), while the application should not have access to it. Offhand, we know of no thread package or operating system that implements such asymmetric threads, and we think this may be an important area of future research, especially when considering the security aspects of such a system.

3.1.3 *Data-Centric Diagnosis*

We also have been working on efficient monitoring of data accesses. Most notably, given the prevalence of the Intel x86 platform, we have been investigating the use of its hardware breakpoint registers as a method for monitoring data accesses.[1] These registers can be used to monitor memory accesses at full CPU speed. They offer a potential basis for deploying efficient monitoring facilities that are data-centric.

Unfortunately, access to these registers is privileged, and the software support must trigger kernel-level traps, and indeed their only use currently is in an observer-process mode (typically, a debugger). Our investigations into using these facilities may lead to suggestions for future hardware or O/S changes, so that self-management can be deployed efficiently and can take advantage of hardware capabilities. This also relates to the idea of a privileged, asymmetric observer thread in a system, where perhaps the breakpoint registers are accessible to the observer thread, while still protected from the application.

Currently, we have not integrated these capabilities with our DDL framework, but we are intending to.

3.2 Supporting Modification and Repair

A framework supporting self-management must not just support monitoring and diagnosis, it must also support the run-time modification of the system. DDL naturally supports this.

The basic capability that DDL offers is the ability to dynamically retarget a binding between shared objects. This simply modifies the built-in jump table that the shared objects use anyways, so no extra overhead is incurred by this basic capability. Since a self-managed system would have diagnosis hooks into the application, it naturally has control points at which it could perform these modifications, or it could use O/S signals as we described in Section 2.

On top of this capability, DDL offers the ability to fully replace a shared object with a new one, by remapping all current bindings from the old to the new one, and then removing the old one. This works under certain constraints—e.g., no indirectly exported symbol pointers, such as a function or symbol address returned from another function. Such bypassing of the "normal" interaction of shared objects is problematic for us to handle, although we will be investigating methods to do so.

Shared object replacement is not the only method for modifying a system, however. DDL can be used to bring in new functionality without removing old functionality. We recently did a case study where we "managed" an architectural simulator to extend its behavior to allow dynamic

[1]Other modern processors also have breakpoint registers, but the x86 line is notable for having four of them.

switching between a detailed (and slow) simulation mode and a partial (but fast) simulation mode [1]. Although we did have to make minor code changes (the original simulator was statically linked and had C "static" (i.e., hidden) symbol declarations), we did not change any functional code but instead used DDL to bring in and dynamically activate the new behavior (fast partial simulation).

Another method for modifying a system is to provide a framework for controlled modification, rather than just simply making the modifications and hoping for the best. Our Hercules framework for supporting an evolution process whereby multiple versions of components can be executed side-by-side is one example of such a framework [3]. In this mode, DDL supports the rebinding of shared object connections to insert framework connections between them, thereby giving the framework the necessary control to perform its duties. We have an initial Hercules prototype working on top of DDL. With dynamic link rebinding, pre- and post-evolution system states incur no extra overhead, since the links are direct, while during evolution, Hercules sits in-between as a complex connector that manages the evolution process.

Our tools do not currently assist in other aspects of modification, such as new component initialization, state migration, and similar issues. These aspects are somewhat orthogonal to the functionality that DDL provides, but we will be investigating them in the future as part of our ongoing work.

4. RELATED WORK

The DITools project [10] 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 [7] 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 [6] 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 [11] 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 [4, 5, 8, 9].

5. CONCLUSION

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, and to make building the infrastructure for this and for self-management features possible.

Our current focus is in the deployment of the Hercules framework on top of DDL, but we are also using DDL for dynamic analysis work (especially scripting language support), for dynamic behavior adaptation, and other applications.

DDL is freely available for research use at http://www.cs.nmsu.edu/please/ddl/index.php.

6. REFERENCES

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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