Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines

Andrew Duncan and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
{aduncan,urs}@cs.ucsb.edu
http://www.cs.ucsb.edu/oocsb

Technical Report TRCS99-09
1 April 1999

Abstract The advantages of virtual machine (VM) execution (dynamically loaded, portable object files with high-level information) also permit changing the semantics of executables. *Load-time adaptation* (LTA) intercepts the VM's file operations and modifies object code on the fly, without changing the VM implementation, without needing access to source code, and without changing the actual files. We introduce a new technique, *library-based LTA*, and show how it can extend languages in such ways as adding contracts or mixins to existing classes, providing default code for interfaces, and instantiating parameterized types. We discuss an implementation of library-based LTA and its application to extending Java semantics.

"Adopt, adapt, improve." — Motto of King Arthur's Round Table

1. Introduction

Programming language researchers are frequently faced with problems that could most conveniently be solved with a small change to the semantics of the language they are using. For example, for Java such changes could include the ability to provide default code for methods in interfaces, mixin classes, assertions, parameterized types, and various kinds of code instrumentation for profiling or debugging. Traditional solutions to these tasks have included source-code preprocessors, design patterns, coding conventions, compiler modifications, and binary code editing. The increasingly widespread use of languages that execute on *virtual machines* (VMs) affords the opportunity to extend the semantics of a language implementation in an efficient and non-intrusive way.

A virtual machine provides a common focus of data and control flow. Instead of compiling all source files for each program into binaries for all platforms, it suffices to compile only the VM itself. This strategy has been used by many popular programming systems for languages from BASIC and Pascal [N+81] through Smalltalk [GR83], ML [L90], Oberon [FK97], Eiffel [SE97], and Java [LY97]. By the same token, the centrality of the VM means that changes at this point can adapt the runtime behavior of the entire language.

We describe *load-time adaptation* (LTA), a code-transformation technique that allows a wide range of changes to programs and languages running on a VM without sacrificing the benefits of portability. LTA comprises a distinctive way of accessing executables: it takes advantage of the common data and control path provided by the VM and intercepts binary files as they are requested by the runtime system. The characteristic benefits provided by VM execution, including binary portability, retention of high-level program information, and dynamic loading of modules, enable LTA to perform its task efficiently and with little disruption of VM behavior. LTA does not require source files for the modified code, so it can operate on system or third-party classes. It also makes no changes to existing class files on disk, so there are no modifications to "back out" of. LTA can defer its work until class loading time, so it affects performance only when active. Finally, it is easy to activate and deactivate the application of changes.

This paper introduces the concept of *library-based* LTA, in which code that modifies program binaries is dynamically linked with the VM. Library-based LTA does not modify the virtual machine's semantics or its implementation, so it does not require upgrading or patching when new VMs are released. We compare this technique to other ways of performing LTA, such as modifying the VM directly, using language-level facilities such as custom class loaders (*e.g.*, subclassing java.lang.ClassLoader in Java), or interceding with the underlying operating system's services. We will describe the intrinsic properties of LTA, common to all implementations, and then compare the properties of different implementation strategies.

We have implemented library-based load-time adaptation for the Java virtual machine (JVM). Our system, called Proteus, lets us change the form of Java class files as they are being loaded. A programmer provides a separate description of the changes to a Java class, and Proteus intercedes between the JVM and the operating system, making corresponding modifications to the class file as it is read from disk or across a network, leaving the original unaltered. The Proteus system embodies a flexible runtime architecture and an extensible front-end, so users can implement custom language extensions. We have used Proteus to implement systems for binary component adaptation [KH97] and for adding contracts to Java classes [DH98]. In addition, we also show how Proteus could add default implementations to Java interfaces, instrument existing code, instantiate parameterized types, and help to work around known hardware or software bugs.

The contributions of this paper are:

- We introduce the technique of library-based load-time adaptation.
- We characterize the range of load-time techniques and describe where library-based LTA fits in.
- We compare the different approaches and explain the advantages of our technique.
- We describe our implementation and show how it can be used to investigate language extensions and to address problems of software engineering and development.

In the following sections we discuss the general implementation and applicability of LTA and our experiences with the Proteus implementation. Section 2 discusses common properties of LTA implementation, and describes the differences between several existing implementations. Section 3 describes our implementation of library-based LTA. Section 4 explains how LTA can accomplish various language extensions, giving details where we have implemented the extension in Proteus. Section 5 describes some prerequisites for applying LTA effectively, and some limitations of LTA. We conclude with a review of related work and conclusions.

2. Load-Time Adaptation

The aim of load-time adaptation (LTA) is to augment a language's semantics or runtime implementation without changing existing tools. To achieve this goal, LTA modifies binary code after it has been requested by a runtime system, but before that system has a chance to use it. Figure 1 shows how a single LTA system can function for more than one runtime system, because it intercedes at a control and data point common to both: the loading of classes. LTA serves as a transparent filter between the VM and its environment: the virtual machine cannot tell that it is not interacting directly with the file system, and the underlying OS can behave as if it is supplying files directly to the requesting process.

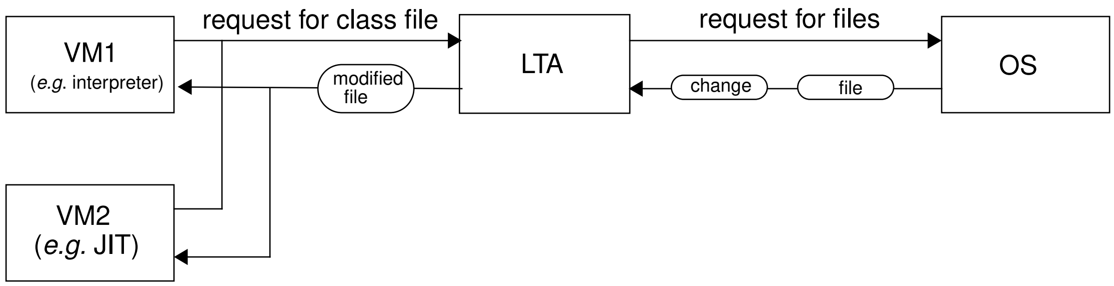

Figure 1. Intercepting a file request with LTA.

There are numerous ways to implement LTA, each with its advantages and disadvantages; we will discuss various alternatives shortly. However, all strategies share several valuable properties that make LTA an attractive option for the kind of language changes we discuss in this paper.

Because LTA intercepts classes at load time, the original class files, whether on disk, network, or even dynamically generated, remain unchanged. This property helps ensure that a project can be ported to runtime systems that do not implement LTA. (Unless LTA adds features essential to the application.) LTA also prevents the profusion of alternative versions of compiled binaries. In addition, LTA simplifies change management: the VM can be assured of having the most up-to-date version of each class file, without having to choose between different modified versions.

In contrast with preprocessor schemes which transform source-level text, load-time interception operates only on class files. This means that existing source code does not need to be recompiled, and there is no need to manage parallel hierarchies of changed and unchanged source text. In addition, LTA can modify system classes or code from libraries for which the source text is not available.

Many VMs implement some form of late or “lazy” loading of classes. In these circumstances, classes are loaded only when necessary. Because LTA defers its processing until a class is actually loaded, it automatically gains the efficiency benefits of late loading—it performs no unnecessary work—without having to address itself directly to the issue of loading policies.

In the following section we will discuss some of the alternatives that have been used in previous projects and compare their merits.

2.1 Strategies for Load-Time Adaptation

The distinguishing characteristic of LTA is the point in the data path at which it intervenes. An LTA engine functions as a filter between the VM and the operating system libraries. Possible ways of intervening at this juncture include modifying the VM’s file-opening calls directly or (for Java) writing a class loader in Java itself, or intercepting the file system operations. Each approach has its advantages and shortcomings.

Modifying the VM. An obvious way to provide load-time filtering is to integrate it into a VM. It is conceptually simple to modify the VM’s file-handling code directly, as shown in Figure 2, adding a stage that modifies class files as they are read from disk or across a network. (If a VM provided a mechanism for

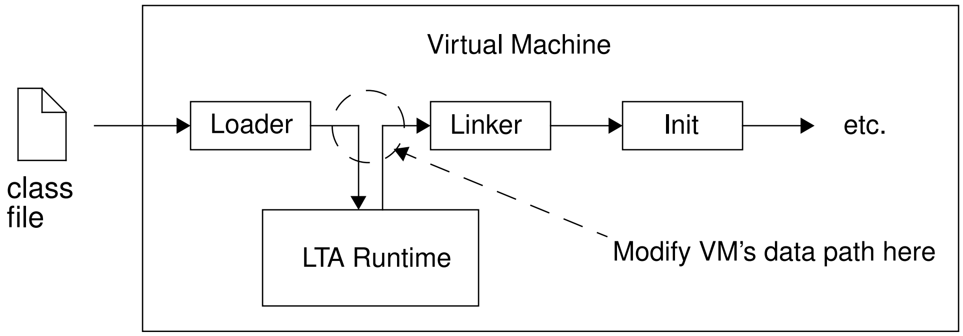

Figure 2. Modifying a VM to implement LTA.

user code to execute at this stage, portable LTA would be considerably easier to implement.) This approach has been used by several groups to augment the Java semantics [KH97, AFM97, MBL97]. Having access to a VM lets a programmer make changes in one place that would otherwise have to be applied to all existing and future code or all compilers for the source language. However, changing the VM itself presents problems: source code for the VM may be undocumented, incomprehensible, or simply unobtainable. Modifying just one VM does not affect the behavior of other runtime systems, thus introducing incompatibilities. But changing all VM implementations is not a realistic goal, and even a single VM may evolve in such a way as to require continuing efforts to keep pace.

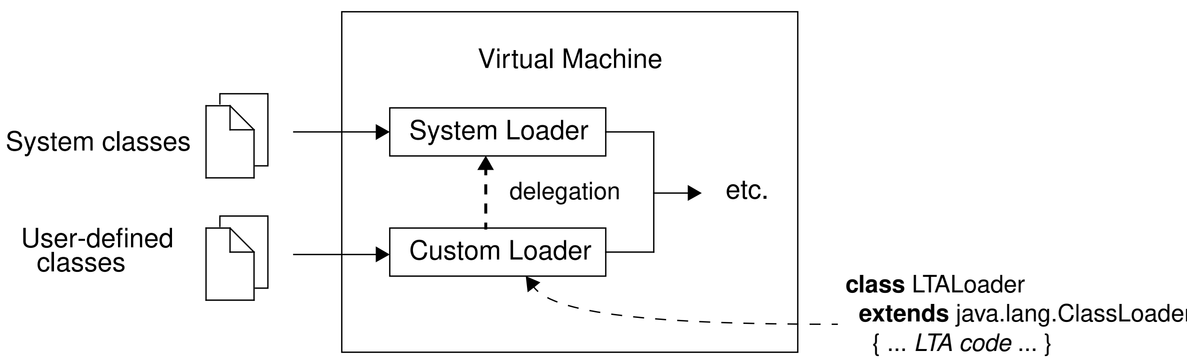

Figure 3. Performing LTA with a custom class loader.

• **Custom class loaders.** The JDK 1.2 specification includes an extension mechanism that allows users to define their own class loaders [LB98]. Class loaders are a consistent and portable scheme for modifying classes during loading [CCK98, WS98]. Figure 3 shows how a user-defined class loader fits into the data path of the JVM. Its implementation involves writing only Java code, hence it is as portable as the language itself. However, custom class loaders have several shortcomings for the task at hand. First, they can only load user-defined classes; all system classes (*e.g.* java.lang.*) are loaded directly by the JVM, so this approach could not modify such classes. Second, applications may use their own class loaders, bypassing the LTA loader that changes the classes. Finally, user-defined class loaders influence the semantics of class accessibility. In an application with two class loaders, both of which delegate to the LTA class loader, class namespaces that were formerly separate will merge, with unpredictable results. This problem is not insuperable, but does highlight the fact that we are not concerned with class loading *per se*, but with having access to the loaded class before any subsequent processing. In the absence of such a built-in hook, we turn instead to the interface between the VM and the OS.

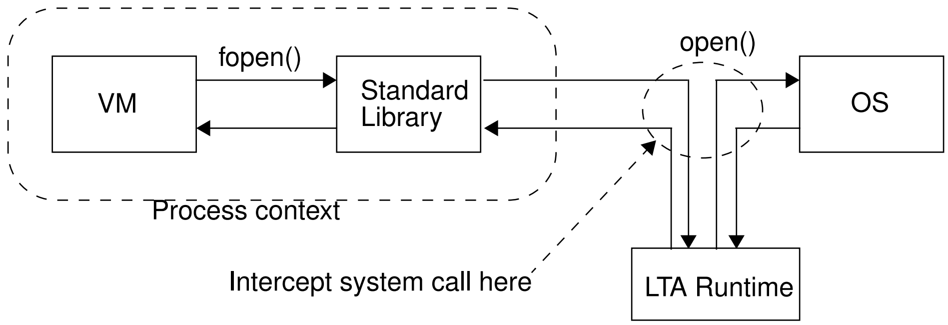

Figure 4. Implementing LTA by intercepting system calls.

• **Intercepting system calls.** The interaction between a VM and the file system is a convenient interception point at which to divert and modify the byte codes being loaded. All operating systems provide an interface through which user processes can request services; such requests are called *system calls*. Figure 4 shows how LTA would be implemented by intercepting system calls. One way to do this is by using the /proc file system available on some versions of the Unix operating system [FG91, AISS98]. Entries in the /proc directory are virtual files that represent the address spaces of running processes, and by manipulating these files a programmer can control the corresponding processes. To use this approach, the VM runs as a child process. The parent process uses the ioctl() call to intercept input and output to the child's virtual file, which are equivalent to system calls by the process.

This approach can implement LTA without changing the actual VM executable itself, and it is possible that the same code could work for several different VM implementations. However, the approach is OS-dependent and thus hard to port to systems without /proc file system functionality. Furthermore, the child process runs in a different address space than the parent and has a different space of file descriptor indices, making it cumbersome to pass data between the interceptor and the VM.

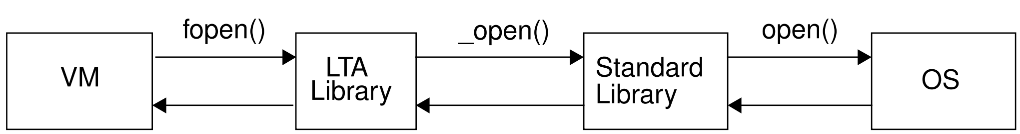

Figure 5. Implementing LTA with dynamic linking.

• **Intercepting library calls.** Instead of using OS-specific ways to intercept file system operations directly, the same effect can be achieved by interposing a custom dynamically-linked library between the VM and the standard libraries it calls. On virtually all operating systems, programs use dynamically-linked libraries (DLLs) to insulate themselves from details of the operating system and hardware they are running on. Such dynamically-linked libraries provide the code for familiar file-handling functions like open(), close(), read(), and so on. These libraries form an intermediate level of services between the user-level process and the low-level details that are specific to the operating system.

By directly modifying existing libraries or by inserting custom variations into a library search path, a DLL-based LTA implementation can intervene during the class loading process (Figure 5). This solution does not depend on OS features that are not uniformly available, while still allowing the application of changes to all incoming files. Because of this portability advantage, our implementation of LTA takes this approach; the next sections discuss library-based LTA in more detail.

2.2 Load-Time Adaptation with Dynamic Linking

Interceding between the VM and the OS at the level of library calls presents a middle path between direct intervention in the VM and source-level custom class loaders. The trade-offs between these two approaches are complementary: altering the VM provides almost arbitrary control to the user at the expense of portability; custom class loaders are as portable as the language itself, but may not be applicable to privileged system classes. Intercepting calls to dynamic libraries maintains for both the VM and the operating system library the illusion that they are conversing directly, and isolates the adaptation engine from the need to know about the origin of the byte codes it is modifying.

The most visible benefit of implementing LTA using a custom library is the decoupling of LTA from a specific VM. This is advantageous for several reasons. First, the developer is freed from the need for privileged access to the VM source code; this would be a considerable restriction in most cases. Second, subsequent changes to a particular VM will probably not necessitate any changes to the LTA code since classes are still loaded using the same system calls. Third, a DLL implementation of LTA will function across different VMs that use the same library interface to the underlying file system, allowing LTA developers to avoid engaging in a race to keep up with a growing variety of virtual machines. For example, the same binary of our SPARC implementation of LTA for Java supports all current Solaris JVMs, from a JDK 1.1 interpreter to the newest JDK 1.2 JIT.

An additional benefit of library-based LTA is the ease with which changes can be activated and deactivated. An LTA runtime library can be inserted into the search path of a VM (or other application) by a simple renaming, by modifying a dynamic search path, or by executing a simple shell script. Reversal of the interception is equally easy. When a VM does not use the LTA library, it suffers no performance penalty.

For Java, library-based LTA avoids several difficulties that arise with user-defined class loaders. To a dynamically-linked LTA engine, all classes are on an equal footing, and can be modified; the JVM uses a special loader for system classes, which are inaccessible to a custom loader. In addition, Java users can define their own class loaders, indadvertently bypassing the LTA engine. If two user-defined loaders both delegate to the LTA loader, the namespaces of their loaded classes will merge, which may change the semantics of an ensemble of classes.

Using dynamically-linked libraries also addresses the issue of coordinating development of new code that relies on modifications to old code. For example, suppose a programmer uses LTA to add a method `m()` to a system class `C`. If this change is applied only at runtime, and not at compile time, no-one will be able to call it—a compiler will reject any calls to `C.m()` in newly-written code, claiming there is no such method. Using library-based LTA, any compiler or other programming tool that requests a class file will automatically get the changed version, without any further intervention on the part of the client.

This approach to load-time adaptation can also apply to executables coming from remote locations, such as web pages or application servers. In such cases, a VM will be making calls to an operating system's network libraries; an LTA runtime engine can intercept these calls as it can calls to file libraries.

In summary, dynamic linking is popular precisely because it allows the flexible replacement of library code without any attention needed from the application developer. This same flexibility is enjoyed by an LTA engine that stands at the crossroads between an application and its libraries, and thus the LTA library can transparently perform its tasks for every virtual machine that links with it instead of with the standard library. Because of the advantages offered by a library-based LTA system, we have used this approach to implement load-time adaptation. The following section describes our implementation in detail.

3. Implementing Load-Time Adaptation for Java

Our implementation of LTA, called Proteus, supplies a modified version of the dynamically-linked standard C library. We add a wrapper to the functions that open a file or return information about it. For example, when the library is called to open a class file, it does its own processing, in the midst of which it calls the C library file functions itself. Proteus provides a small set of elementary transformations that can be applied to a class file; in combination these changes can have wide-ranging effects. We discuss the Proteus runtime engine, which is the core of the LTA implementation, and the front end that allows users to specify the changes to be applied to classes.

3.1 The Runtime Engine

Figure 6 shows some of the control and data flow for modifying a class. In the figure, the VM (shown at left) calls the open() library routine to start loading a class. Proteus's version of open() reads the class file, and a *delta file*, by calling the C library's original open() function. Proteus parses the class file into an internal representation, applies the changes specified in the delta file, translates the result back into class file format, and writes the modified class file to a user-specifiable cache location on disk. Finally, Proteus passes the new path to the C library's file-opening routine and returns the resulting file descriptor.

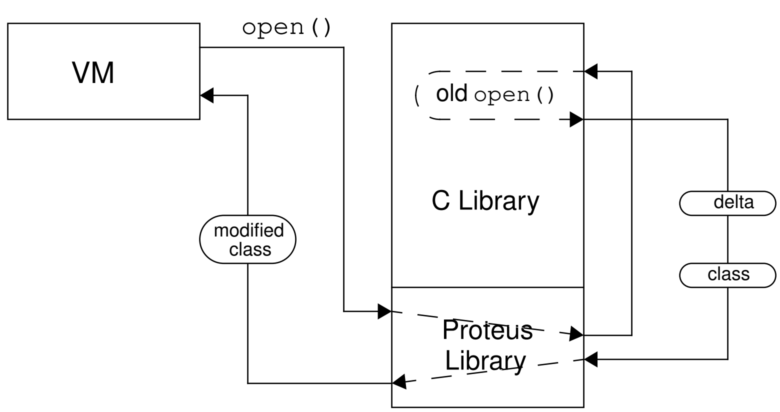

Figure 6. Data and control flow in the Proteus runtime engine

When the Proteus runtime engine detects a request to open a class file, it first looks in its cache directory to see if the class has already been modified and stored. If so, Proteus just redirects the system call to the cached version and passes the call to the C library. If not, Proteus looks for a delta file first in the same directory as the class file, then in the cache directory. Looking in the cache directory facilitates specifying changes to classes (*e.g.* system classes) whose directories are not writable by the Proteus user. If there is a delta file for the class, Proteus applies the changes to the class file, caches it and redirects the system call to the new file. If there is no delta file, Proteus has no effect, just passing the call unchanged to the C library. When Proteus locates a cached version of the class file, it also checks to see that the class file is up-to-date with respect to its contract file and the original class file. If the class file is out of date, the library reapplies the delta file to the class and caches the result as described above.

The standard C library implements many file-related routines. For example, the Solaris 2.6 library libc.so.1 contains functions open(), _open(), and several others. To make the Proteus runtime engine work with all VMs on a given platform, we modify each file routine to check for class files. In addition to open() and variants, we also instrument the stat() call which returns information about the file, in particular its size. All changed functions follow the process described above, looking for the delta file and modifying the class file if appropriate.

The procedure for creating the Proteus runtime engine is similar for most platforms: edit an existing library and compile replacement code. The two resultant modules may remain separate or be linked together. For example, to create a version for a Unix platform, we first make a working duplicate of the dynamically-linked C library libc.so.1. We then edit this binary file directly, renaming the functions we want to intercept, *e.g.* open() becomes OPEN(). We create a new source file redefining open() to carry out the process of adding contracts to classes, itself calling the renamed C library version as necessary. Proteus also incorporates part of the BCA runtime library [KH97, KH98] to perform class file modifications. Finally, we compile and link this code with the existing C library, resulting in the Proteus runtime engine. The Proteus engine consists of about 8,000 lines of C++ code in 25 classes.

Because the Proteus runtime engine is dynamically-linked, it suffices to arrange that the off-the-shelf JVM finds and uses it in place of the standard C library. Usually, a simple shell script setting the library search before starting the JVM is all that is needed to enable LTA for a new JVM.

Our current implementation for Solaris 2.6 supports all Solaris JVMs (*e.g.* the JDK 1.1.x interpreters, the JDK 1.2 interpreter, and the Solaris 1.2 JIT); these JVMs differ slightly in the way they open class files but are all supported by the same library. We are currently porting our code to the Win32 platform and its popular JVMs.

3.2 The Front-End

The purpose of the front-end is to generate the delta file, which specifies the changes made to a class file by the runtime engine. The delta file describes these changes in a binary form suited for efficient runtime application; the front-end is a compiler that lets the user specify the changes in human-readable form. The compiler is written in Java using a parser-generator and comprises about 100 classes.

The syntax of specification and delta files is borrowed from the Keller-Hölzle implementation of binary component adaptation [KH98]. As an example, the specification for adding some guard code to a stack implementation might look like this:

```
adapt class Stack {
  rename method void push(int i) to push$old;
  add method public void push(int i) {
    if (is_full()) throw new RuntimeException("Stack is full in Stack.push().");
    push$old(i);  // Call original version.
  };
}
```

A user can specify renaming class or instance members or references to them, or adding members, and carrying out these changes to specific classes or interfaces, or all implementors of an interface. In this way the specification language provides the elementary building blocks of class modification. Because Proteus is currently concerned with adapting the interface (in the general sense) of a class, it does not provide for direct byte code manipulation, but this is a possibility for the future.

In addition, Proteus supports the addition to the compiler of other syntaxes, to allow users to develop custom specification languages for specific purposes. For example, a Proteus user might prefer to describe the modification to the Stack class above in this form:

```
guard Stack.push(int i) with { !full() else "Stack is full in Stack.push()." }
```

To this end, Proteus provides an API for integrating a custom parser into the existing delta file compiler. In addition, the internal syntax trees used by the compiler are equipped with methods that will print their reverse-compilation into the standard textual specification format, so it may conveniently be inspected just as if it had been generated directly. The Proteus compiler also supplies an interface for applying the Visitor design pattern [GOF95] so a client can easily manipulate the tree before it is converted to the delta format.

In the interest of simplicity, Proteus currently requires one delta file per class. However Java supports the combination of many class files into a single *jar file*. It would not be difficult to extend Proteus in a similar way, for example by allowing one compound delta file per jar file.

4. Applications of Load-Time Adaptation

An LTA system can perform many useful language extensions. Among these are modifying classes to adjust the fit between libraries and clients; adding language features such as assertions, default implementations for interfaces, or parameterized types; instrumenting code for measurement and analysis; and applying patches or workarounds for known bugs. We have constructed several of these tools using Proteus. In the following sections we describe the motivation for such modifications, discuss how LTA simplifies their implementation, and describe in more detail those systems we have implemented.

4.1 Binary Component Adaptation

Binary component adaptation (BCA) [KH97] was conceived as a remedy for the problems posed by the separate development of code modules [H93]. Although object-oriented (OO) programming paradigms have taken some steps toward the easier reuse and evolution of software components, it is still easy for a client to be stymied by a small incompatibility between classes. Examples of such problems include differences in parameter order, mismatch of class member names, and classes that implement interfaces but don't declare that they do.

Clients will not usually have access to the source text of the incompatible code. Furthermore, making changes to one offending class or library will not help when another similarly problematic body of code is dynamically loaded. In such cases what the client wants is a way to change how a category of classes appears to another class or to the runtime system, rather than the ability to make arbitrary changes to the code.

The original Keller-Hölzle BCA implementation directly modifies the JDK 1.1.5 Java VM to apply the desired changes to a class file, and applies BCA to the javac compiler itself to ensure that new classes are compiled against modified old classes. This restricts the client to use a particular Java implementation. By re-implementing BCA using the Proteus LTA mechanism, we have eliminated the dependency on a particular implementation of the JVM or Java compiler.

4.2 Contracts

Contracts [M88] form an extension of the type constraints already imposed by a class's interface. A contract specifies conditions that must apply on entry or exit to a method (pre- or postconditions), or that must hold generally (class invariants). Various researchers have suggested ways of adding contract support to existing languages that lack them [Kr98, KHB98, MP98, PSS98, C98]. We have used the Proteus framework to implement Handshake [DH98], a system for adding contracts to Java classes.

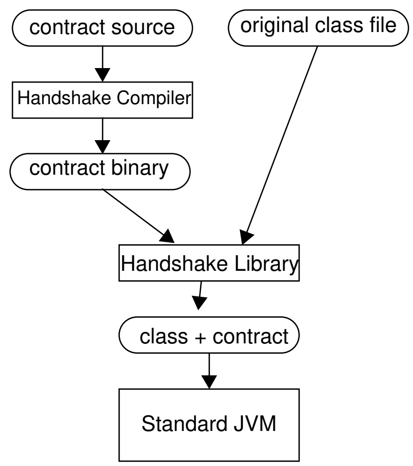

Figure 7. Adding a contract to a class with Handshake.

The Handshake system comprises a contract compiler and the runtime library. Figure 7 shows how the original class file and the assertions are merged at runtime. To add a contract to a class with Handshake, a programmer creates a *contract file* which is associated with a single class. For example, a contract for a Stack class could, in part, look like this:

```
contract Stack {
  invariant size() >= 0 else "size is negative";
  public void push(int i)
    pre !is_full() else "stack is full" ;
    post top() == i;  // Error message is optional.
}
```

The Handshake front-end uses the API provided by Proteus to translate this into a delta file representing changes. The Proteus runtime will modify the Stack class as shown in Figure 8.

To evaluate the overhead imposed by contracts, we added empty contracts to 107 classes of javac, the Java compiler provided in the JDK. We measured the performance using JDK 1.2beta4, on a Sun Ultra-1/170 167MHz workstation running the Solaris 2.6 operating system.

The unmodified compiler took 3.13 seconds to compile a "Hello World" program. The added time required to divert compiler's file-handling system calls through the Handshake runtime library was less than half a second. Processing the classes— finding the contract file, ascertain that there is no cached class, read and parse the class into memory, perform the modifications (in this case none), un-parse and write the file to the

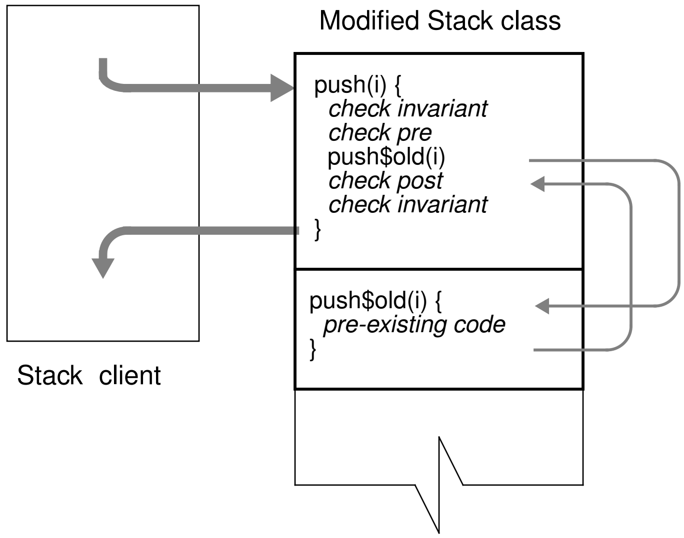

Figure 8. Renaming and wrapping the `push()` method.

cache—took an additional three-quarters of a second. (So even in this worst case the performance penalty for total running time is only about 1/100th of a second per instrumented class.) Subsequent compiles, when the processed class files were cached, took only 3.5 seconds, again less than half a second slower than without using Handshake.

4.3 Interface Mixins

The Java language provides an interface construct, that lets a programmer specify a subtyping relationship without committing to an implementation. A programmer wanting to extend a pre-existing interface will face a problem: pre-existing implementors of the interface may no longer link successfully because they do not implement the newly-added methods. Load-time adaptation can solve this problem by ensuring that all implementors of the interface are supplied with the appropriate methods. This has the effect of providing with an interface some default implementation, and of bridging the gap between Java-style single inheritance and multiple inheritance in the style of C++ or Eiffel.

The provision of default methods for interface types is a special case of the concept of a *mixin* [BC90, B92]. In general, a mixin represents the difference between a class and a subclass, abstracted from a connection to any specific class. A mixin can easily be expressed and implemented using LTA. Consider the case of a graphical object, for example, an interface like this:

```
interface GraphicalObject {
   DrawingEnvironment env();
   void draw();
   // etc...
}
```

Suppose want to add to such an object the concept of a bounding rectangle, outside of which no image will appear when the object is drawn. A mixin specification for this behavior could look as follows:

```
mixin BoundRect mixes with GraphicalObject {
   int left, top, right, bottom;
   void setRect(int l, int t, int r, int b) { ... }
```

```
    void draw() { env().setClipRect(left, top, right, bottom); super.draw(); }
}
```

Although the mixin specification appears to describe a new inheritance relation, it will be implemented as a code-copying process. LTA will modify only those classes that directly implement GraphicalObject; and any further derived classes will remain unchanged. However, dynamic dispatch of method calls ensures that the modified draw() method will be in place for all subclasses of the top-level implementor. (Of course, if subclasses choose to override draw() completely, not calling their parent's version, they will not receive the benefit of the mixin.)

It would be possible to provide similar mixin functionality without using LTA. For example, a user could modify the Java compiler to automatically insert this code whenever compiling a class that implements GraphicalObject. However, this constrains the user to Java compilers for which the source is available. LTA can accomplish the same task in a less intrusive way. In addition, LTA provides additional functionality since it enables dynamic mixins, i.e., allows run-time (load-time) mixin application. Therefore, mixins can apply to pre-existing classes or even to future classes (such as new classes that implement GraphicalObject). That is, dynamic mixins apply to an open set of classes, and the mixin writer does not need to know the complete set of classes to which the mixin should be applied.[1]

4.4 Parameterized Types

In the previous section we described a form of parameterized inheritance, in which code is added to classes that satisfy some constraint, such as implementing a given interface. In that case, we think of the mixin as a distinct entity, and the parameter is the class with which it is mixed. In a similar manner, a *generic type* is a partially-defined type, that needs a parameter (which is another type) to complete the definition. Such a construction allows a programmer to specify an algorithm or data structure in a way independent of the objects contained or manipulated. Support for parameterized types takes such forms as Eiffel generics [M91], ML polymorphic data types and functions [Wi87], and C++ templates [PSLM96]. Java does not directly support parameterized types; various researchers have proposed mechanisms for extending Java to allow some form of genericity [MBL97, OW97, AFM97, BOSW98, CS98].

Several of these approaches require customizing the class loading process to instantiate the generic type. For example, in the system of Agesen, Freund, and Mitchell [AFM97], a custom Java compiler translates parameterized classes into conventional Java byte codes with additional annotations. When compiling a parameterized class Stack<T> the resulting generic class file uses a recognizable placeholder name, such as $1, for all references to the formal type parameter T. When compiling code that declares a stack instance like Stack<Window> the compiler rewrites the type name in a distinctive way, such as Stack$Window.

When the runtime system encounters a reference to Stack$Window, it must recognize the class name as representing a generic class that needs further processing. The authors' system uses a custom class loader [LB98] to recognize the names, extract the names of the parameters, and substitute these into the generic byte codes to create the specific class. However, the authors found that they were unable to make their custom class loader the default loader—and so be able to filter system classes as well as user classes—without making changes to the VM itself.

Load-time adaptation can apply the requisite changes in precisely the same way, but without the trade-off of either altering the VM or giving up the ability to modify system classes. The original class files, along with the added annotations, contain all the information necessary to instantiate the generic type to a desired

[1] This is in contrast to the more conventional static view of mixins as building blocks for constructing new classes from scratch. We view dynamic mixins not as a replacement to static mixins, but as a complement.

specific one. Since an LTA engine has access to all incoming class files, it can assure that all necessary processing takes place before the VM ever sees a byte code.

4.5 Code Instrumentation

Most programming language systems and platforms provide some support for measuring and analyzing the performance of programs. Often this takes the form of compile-time options, instructing the compiler to insert instrumentation code at entry and exit to functions, methods, or basic blocks. This approach requires access to the source code of the programs or modules to be measured. An alternative is to modify executables and object modules on disk. For example, Purify [HJ92] modifies existing binaries by inserting instructions to check memory accesses. Lee and Zorn [LZ97] have implemented the Bytecode Instrumentation Tool (BIT) for instrumenting Java byte codes with profiling tools by making changes to the existing class files on disk. Other examples include QPT [LB94] and Epoxie [Wa92].

These methods all modify existing object code. In this case, the user needs to run a tool applying the changes after every recompile. This is added work for the programmer, who must keep track of out-of-date classes and change specifications, and possibly maintain a separate repository of modified classes. By contrast, load-time adaptation can accomplish the same goals without multiplying the number of existing versions of a class or object module. Off-line techniques do enjoy the advantage of whole-program visibility; although a load-time engine can cache information about all classes that have gone through its bottleneck, it will be difficult to reason about as yet unloaded code. But conversely, off-line techniques require that the entire program be known before runtime; they do not deal well with languages (such as Java) that support dynamic linking. Load-time adaptation is not only compatible with late linking, it may be thought of as part of the very process.

A similar application of LTA is code instrumentation for debugging. Lencevicius *et al.* [LHS99] have developed a system that modifies classes by adding debugger invocations for object creations and field assignments. Classes are modified at load time with a user-defined class loader [LB98]; as discussed above, this precludes instrumenting system classes, a limitation that could be avoided with library-based load-time adaptation.

A related application of LTA instrumentation would be to add code that checks parameters to API calls. This is similar to implementing contracts, but works exclusively in the client code. An LTA pass could inspect incoming byte code, recognize calls to specific libraries, and add code to test the parameters for desirable properties.

4.6 Dynamic Code Rewriting for Bug Workarounds

Even programs that are textually correct may produce runtime errors due to bugs in compilers or lower levels of the execution environment such as system libraries or hardware. Load-time adaptation can provide a minimally intrusive workaround for such problems. For example, in 1994 Intel released a version of the Pentium CPU with a flaw in its floating-point division implementation. The hardware division algorithm for 64-bit numbers would return an incorrect result for certain combinations of divisor and dividend. The set of ratios that would trigger this error was quite small, as was the error itself, but repeated exercise of this flaw could gradually accumulate the error. In most situations it was impossible to predict whether or at what steps the erroneous results would appear, so *a priori* modification of the algorithm would not help.

One solution to such a problem is to modify existing compilers to emit code that checks for divisors and dividends that trigger the bug, and redirect execution to a slower but more accurate division algorithm. This approach requires coordination between many independent vendors and researchers, and requires users to replace all affected applications with recompiled versions. A better solution would edit executables in-place, looking for (and rewriting) the problematic instructions. Wolfram Research [Wo94] released a program that acted as a wrapper, processing and patching an executable before running it.

Load-time adaptation can provide an even more convenient solution for such situations. As a class is being loaded, an LTA engine can examine each byte code, detect sequences of instructions that may trigger a bug in external code, and rewrite the byte code to avoid the problem. These changes would be applied only when necessary; for example, the same class file (downloaded from a file server) may be rewritten on one client machine but not on another client with a later (more correct) floating point implementation.

5. Limitations of Load-Time Adaptation

The applicability of load-time adaptation may be hampered by some properties commonly found in language systems that do not use a virtual machine. Although VM execution is not a requirement for using LTA, most VMs have in common a design that facilitates load-time adaptation. Virtual machines typically execute byte codes that are not specific to a real hardware platform. For example, the structure of an object file may vary widely across different hardware platforms, making a general-purpose runtime engine difficult to create. VM-based languages such as UCSD Pascal [PV84], Smalltalk [GR83] and Java [LY97] use a byte code format that represents an instruction sequence; the byte codes define the architecture of a machine (usually implemented in software) on which they will run. Although it is possible to apply LTA to languages without a uniform executable format, this property makes implementation much more practical: the LTA engine need not be rewritten, only recompiled.

LTA relies only on the existence of a VM-usable portable object code format, not its precise form. For example, LTA could readily be applied to a language implementation like MacOberon [FK97], which uses a compressed syntax tree for its executable code. By contrast, the TDF project [DRA93] also uses a architecture-neutral tree-structured representation of object code, but programs are linked together before runtime into a native code format. In the absence of a virtual machine that uses the portable tree-based code as-is, LTA cannot easily be applied.

Binary portability *per se* is not sufficient to allow for the efficient implementation of LTA. If the class files no longer retain any high-level information about the program they represent, an LTA engine will be forced to expend considerable effort to recover it. Such information is often present in byte code formats. In order to support type-safe dynamic linking, Java byte codes retain type information about fields and methods. The tree-based binaries of MacOberon embody the control-flow topology of the original program, making much easier the runtime application of many optimizations [KF97]. Of course, the specific sort of information in the byte code will determine what kind of load-time modifications are reasonable.

In addition, the process of loading object files must be accessible to the LTA engine. For statically-linked native executables, regardless of platform, the loading of object modules is effectively out of reach. Even if a program uses dynamically-linked libraries, if their loading is managed by an OS-specific process, it will be (at least) difficult to intervene. By contrast, a VM runs as a regular user process, so its file system operations go through a well-known interface. This provides various opportunities for reliably intercepting and modifying those operations.

Intervening in the process of file loading requires some path of access. An ideal way to process class files at load time is to be given explicit but constrained access by the VM. This differs from the custom class loader approach, which allows the programmer to redefine the meaning of loading a class; by contrast all that LTA requires is that it be given read/write access to an array of bytes at a clearly defined stage of class loading. If the VM presents such an interface in the language it interprets, the LTA phase will enjoy the same portability as any other source code; this is also an advantage of the custom class loader approach.

Similarly, load-time adaptation may have difficulty changing code that is dynamically generated—that is, created in binary form by a running program and then executed; in such a case there may be no file-handling steps at all. This situation can also be addressed if the language's runtime system provides a hook for external procedures to inspect and modify the dynamically generated code before it passes through subsequent validation and linking stages.

It is possible to use LTA to make changes to a class that invalidates it, either intrinsically or as it relates to other classes. A program transformation is called *binary compatible* if it leaves unchanged the ability of pre-existing binaries to link with it [DWE98]. We would like to characterize the binary compatibility properties of these transformations, both separately and in combination. In addition, other properties of these transformations under composition are not yet well understood. Beyond the concept of link safety, program designers need to concern themselves with semantic compatibility. It is possible to make changes to a class that do not cause linking errors, and that appear to conform to reasonable OO design practice, but which will have unintended runtime behavior in the presence of unexpected changes to the class's descendents [MS98]. We would like to examine the semantic compatibility of the transformations made possible by LTA.

6. Related Work

Binary Component Adaptation [KH97, KH98] and the parameterized type systems of Agesen, Freund, and Mitchell [AFM 97] and the Thor group [MBL97] all use direct modification of a virtual machine to extend program semantics with load-time processing. As we noted above, this approach sacrifices the portability benefits provided by the VM environment.

The Java Object Instrumentation Environment (JOIE) [CCK98] provides a framework for constructing transformations of Java classes. Its operations include not only interface modifications, such as field and method addition and renaming, but also direct modification of byte codes. Wallach and Felten [WF98] have used JOIE to modify byte codes to enhance the Java security model. JOIE has access to the class files at load time through a custom class loader [LB98]. This has the advantage that the entire project is written in Java, hence is fully portable, but the user cannot instrument system classes using JOIE.

Dalang [WS98] also uses a custom class loader to gain access to Java class files as they are being loaded. (The authors also note the desirability of a standard JVM hook for processing class files between physical loading and subsequent processing.) It uses the reflection features of Java to generate for each loaded class a wrapper class with the same interface; the wrapper inherits from a metaobject class that defines operations to be performed before or after method invocations.

Lee and Zorn have implemented the Bytecode Instrumentation Tool (BIT) [LZ97], a collection of Java classes that modify Java byte codes. This system does not address the issue of how to get access to the byte codes; it could be used by any of the LTA approaches discussed here. The ClassFile package [D98] is a similar set of tools. The Darwin project [Kn98] uses this package to add prototype-style inheritance to Java.

7. Conclusions

We have described the implementation and application of load-time adaptation (LTA), a flexible approach to post-compilation processing of executable code. LTA intercepts and modifies class binaries as they are being loaded at runtime and leaves the original files unchanged, allowing it to apply to classes for which source code is not available. By performing work only when classes are loaded, LTA automatically shares the efficiency benefits of a runtime system's dynamic loading policy.

Virtual machine execution has several properties that make LTA implementation more practical: virtual machines run as ordinary user processes so file system operations are more easily accessible for interception; they use portable executable formats, so modification code can itself be portable; and their high-level byte codes facilitate high-level reasoning about and manipulation of program binaries. Because of these properties, LTA for virtual machine-based languages can provide uniform support for language extension across many different implementations.

We have compared the properties of implementing load-time adaptation by changing the virtual machine directly, by using custom class loaders defined at the source language level, and by intercepting file system calls. Using dynamically-linked libraries to intercept file system calls has the advantage that the LTA code can be made independent from specific virtual machine implementations. Library-based LTA inserts an additional processing layer between a virtual machine and the operating system, forming a gate through which all file operations must pass. All programming tools—virtual machines, compilers, analysis tools— that use a standard library to call the underlying operating systems will be identically affected by library-based LTA. Such an implementation will also be easy to activate and deactivate. Library-based LTA exacts only a modest performance penalty: the only overhead typically comes from an additional file lookup each time the running program asks for a class. This extra time represents a small fraction of the total running time of any substantial program.

We have implemented a version of LTA for the Java Virtual Machine, and used it to explore some applications of LTA to extending the behavior of Java programs. We can modify classes to smooth the integration of library classes with clients, add Eiffel-style contracts to Java classes, and add default implementation to Java interfaces. In addition, LTA can perform code transformations such as instantiating parameterized types, generating new classes via mixins, instrumenting code, and dynamic code rewriting, that have traditionally been performed by other means. We believe that load-time adaptation can provide the framework for investigating future extensions to languages running under a virtual machine, as well as making existing code easier to run and new code easier to write.

Acknowledgments

This work was funded in part by Sun Microsystems, the State of California MICRO program, and by the National Science Foundation under CAREER grant CCR96-24458 and NSF grant CCR-9804061. Many thanks to Ralph Keller for his extensive work on binary component adaptation. The authors are grateful to Jeff Bogda, Karel Driesen, and Sylvie Dieckmann for their helpful discussions and comments.

8. References

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. *Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 1997.

[AISS98] Albert Alexandrov, Maximilian Ibel, Klaus Schauser, and Chris Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. *Proc. of the USENIX 1997 Technical Conference*, January 1997.

[B92] Gilad Bracha, *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. Thesis, University of Utah, 1992.

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 1990.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. *Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1998.

[C98] Igor D. D. Curcio. ASAP - A Simple Assertion Pre-processor. *ACM Sigplan Notices*, December 1998.

[CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic Program Transformation with JOIE. *Proc. of the 1998 USENIX Annual Technical Symposium*.

[CS98] Robert Cartwright and Guy L. Steele Jr. Compatible Genericity with Run-time Types for the Java Programming Language. *Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1998.

[D98] Markus Dahm. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik. Available at ftp://ftp.inf.fu-berlin.de/pub/JavaClass/report.ps.gz.

[DH98] Andrew Duncan and Urs Hölzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, 1998.

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java Binary Compatibility? *Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1998.

[FG91] R. Faulkner and R. Gomes. The Process File System and Process Model in UNIX System V. *Proc. of the Winter 1991 USENIX Technical Conference*, January 1991.

[FK97] Michael Franz and Thomas Kistler. Slim Binaries. *Communications of the ACM*, December 1997.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Software*. Addison-Wesley, 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[H93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. *Proc. of the European Conf. on Object-Oriented Programming*, Springer-Verlag, 1993.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. *Proc. Winter USENIX Conference*, January 1992.

[Kn98] Günter Kniesel. Type-safe Delegation for Dynamic Component Adaptation. ECOOP 98 Workshop on Component-Oriented Programming. Available at http://www.abo.fi/~Wolfgang.Weck/WCOP/98/Papers/Kniesel.ps.

[Kr98] Reto Kramer. iContract — The Java™ Design by Contract™ Tool. Available at http://www.promigos.ch/kramer.

[KF97] Thomas Kistler and Michael Franz. A Tree-Based Alternative to Java Byte-Codes. Technical Report No. 96-58, Department of Information and Computer Science, University of California, Irvine, 1996. To appear in *Proc. of the Intl. Workshop on Security and Efficiency Aspects of Java '97*.

[KH97] Ralph Keller and Urs Hölzle. Binary Component Adaptation. *Proc. of the European Conf. on Object-Oriented Programming*, Springer-Verlag, July 1998.

[KH98] Ralph Keller and Urs Hölzle. Implementing Binary Component Adaptation for Java. Technical Report TRCS98-21, University of California, Santa Barbara, 1998.

[KHB98] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: Reflective Java Library to Support Design-by-Contract. Technical Report TRCS98-31, University of California, Santa Barbara, 1998.

[L90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990. Available at ftp.inria.fr.

[LB94] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software — Practice and Experience*, February 1994.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1998.

[LHS99] Raimondas Lencevicius, Urs Hölzle, and Ambuj Singh. Dynamic Query-Based Debugging. To appear in *Proc. of the European Conf. on Object-Oriented Programming*, Springer-Verlag, 1999.

[LY97] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1997.

[LZ97] Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. *Proc. of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[M88] Bertrand Meyer. *Object-Oriented Software Construction (1st Ed.)*. Prentice-Hall, 1988.

[M91] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[MP98] Mike Mannion and Roy Phillips. Prevention is Better Than a Cure. *Java Report*, September 1998.

[MS98] Leonid Mikhajlov and Emik Sekerinsky. A Study of the Fragile Base Class Problem. *Proc. of the European Conf. on Object-Oriented Programming*, Springer-Verlag, July 1998.

[N+81] K.V. Nori, U. Ammann, K. Jensen, H.H. Nageli and Ch. Jacobi. Pascal-P Implementation Notes. In *Pascal—The Language and its Implementation*, D.W. Barron, ed. Wiley, 1981.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[PSLM96] P.J. Plauger, A. A. Stepanov, M. Lee, and D.R. Musser. *The Standard Template Library*. Prentice-Hall, 1996.

[PSS98] Jeffery E. Payne, Michael A. Schatz, and Matthew N. Schmid. Implementing Assertions for Java. *Dr. Dobb's Journal*, January 1998.

[PV84] Daniel R. Perkins and Dennis Volper. UCSD Pascal on the VAX, Portability and Performance. *Software — Practice and Experience*, May 1984.

[SE97] SmallEiffel versions -0.83 and later. Available at http://www.loria.fr/projets/SmallEiffel.

[Wa92] David W. Wall. Systems for Late Code Modification. In *Code Generation — Concepts, Tools, Techniques*, ed. Robert Giegerich and Susan L. Graham. Springer-Verlag, 1992.

[Wi87] Åke Wikström. *Functional Programming in Standard ML*. Prentice-Hall, 1987.

[Wo94] Wolfram Research Inc. Pentium FDIV patcher available at http://www.mathsource.com.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. *Proc. of the 1998 IEEE Symposium on Security and Privacy*, May 1998.

[WS98] Ian Welch and Robert Stroud. Dalang—A Reflective Java Extension. *Proc. of Workshop on Reflective Programming in C++ and Java*. UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan, ISSN 1344-3135, October 1998.