March 7, 2008

RT0781

Computer Science 10 pages

Research Report

Bytecode Transformation-Tolerant Bytecode-Transformation in Java

Michiaki Tatsubori

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Limited Distribution Notice

This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

Bytecode Transformation-Tolerant Bytecode Transformation in Java

[Extended Abstract]

Michiaki Tatsubori
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
mich@acm.org

ABSTRACT

Bytecode transformations at class-loading time in Java offer a good trade-off point in terms of dynamicity and power of transformation, execution performance, and portability of the applications transformed. However, they often limits the use of custom/system class loaders in application programs. This limitation is especially the pain for developing stacked containers, which are often used in application servers. This paper presents a transformation framework called EMPL for addressing this limitation. A transformation container using EMPL allows its application software stacks to transform the bytecode of their interest using customized class loaders. This paper also presents how to implement EMPL on top of the existing standard Java virtual machine and system class libraries without modifying them.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, software libraries*

General Terms

Algorithms, Design, Languages, Experimentation

Keywords

Generative programming, bytecode transformation, Java class loader

1. INTRODUCTION

The customizable class-loading mechanism in Java [6] provides the language with great flexibility. The first-class class loader objects allow customizing class-loading behavior by assigning the responsibility for class loading to designated class loaders, including user-defined class loaders [13]. Especially, techniques with bytecode transformation in a class loader are getting popular these days for providing *transformational systems* [11], which is middleware that rewrites overlaying application programs for the support of functional extensions. They are popular because of the good trade-off point offered by the techiniques in terms of dynamicity and power of transformation, execution performance, and portability of the transformational systems and the application programs transformed [3, 4].

SIGSOFT 2008/FSE 16 November 9-15, 2008, Atlanta, GA, USA

However, program transformations face serious restrictions when class loaders are used as first-class objects in an application program, which is the target of the transformations. In fact, existing transformational systems implemented as customized class loaders often evade grappling squarely with this situation. They disable these features when they find any use of reflection in an transformed program or, in the worst cases, their manuals simply state that any use of customized class loaders unpredictable results.

It is desired that application programmers can benefit from the flexibility of the customizable class loader mechanism in their programs while still allowing the programs to be supported by middleware platforms for functional extensions like persistency, distribution, high performance, good reusability, and so on. Unfortunately, current solutions do not offer both benefits since it is not possible to combine customized class loaders in application programs and transformation by a middleware platform. However, both of these capabilities are important for programmers. On the one hand, customized class loaders are common and essential techniques for a wide range of programs with dynamic behavior. On the other hand, program transformations are essential techniques for supporting extended functions that crosscut application programs and which cannot be provided in the usual class libraries.

Existing proposals that achieves the mandatory transformation of application programs require modification of Java virtual machine (JVM) or bootstrap class libraries. Such examples would be Binary Component Adaptation (BCA) [7] and JMangler [10], both of which enables compiled Java classes to be modified during load-time. With BCA, modifications are declared in a special language that offers a predefined set of transformations. It modifies a JVM implementation to hook into the class loading process. It is tied in a specific platform and a specific JVM version. JMangler only modifies bootstrap class libraries so it is portable to a platform where bootstrap class libraries are written in a pure-Java manner. Unfortunately, JVM vendors often prohibit to re-

place bootstrap classes through some legal restriction. For example, doing so would contravene Sun's binary code license for JVM (the Java 2 Runtime Environment binary code license).

This paper addresses this problem with a framework called EMPL. This framework:

- makes program transformations by middleware mandatory and transparent for the modified application programs so that the transformation by the middleware is always ensured to be performed on the application programs, and
- runs on the regular JVM without modifying its implementation or its bootstrap class library.

The design and implementation of EMPL described in this paper is built just as a customized class loader thus it can be run on regular Java virtual machines.

The rest of this paper is organized as follows. First, we use an example scenario to clarify the problems that a transformation container faces when program transformation is used in application programs. In Section 3, we introduce our framework for addressing this problem and present the design and implementation of the framework called EMPL. Section 4 shows applications of EMPL and some experimental results of measuring overhead with EMPL. We finally conclude this paper and talk about future directions of this work in Section 5.

2. TRANSFORMATION CONTAINER MIDDLEWARE

The high dynamism provided by reflection is a thorn in the side of program analysis/rewriting technologies. This section gives an example scenario of a middleware platform that transforms the overlaying application programs to provide them with extended functionality. By showing example problems with this scenario, we clarify the problems that a transformation container faces when reflection is used in application programs.

2.1 A Motivating Example Scenario

Suppose we want to provide application-scoped properties instead of system properties. System properties use system-wide key-value pairs for specifying parameters of the system behavior. Instead, we want to provide properties scoped only in each application program. For example, the following code for the static method invocation of `setProperty()` in `java.lang.System` sets a system property in Java.

```
System.setProperty(
    "java.xml.sax.driver",
    "org.apache.xerces.parsers.SAXParser")
```

This expression binds a value "org.apache.xerces.parsers.-SAXParser" to the key "java.xml.sax.driver" in order to specify that an Apache implementation should be used when the standard abstract XML parser API is used in the program.

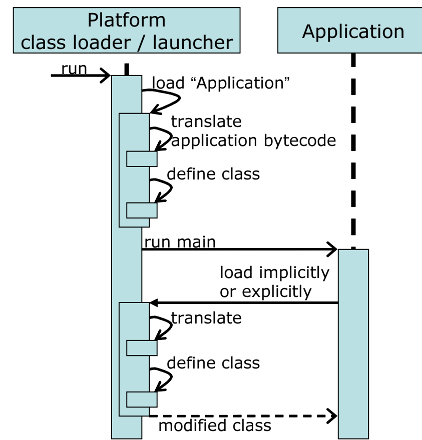

Figure 1: A sequence diagram of typical load-time bytecode rewriting. A class loader performs bytecode transformation when it is either explicitly invoked to load application classes from the platform launcher, or is implicitly invoked to resolve required classes in the application program.

To prevent this kind of application-setting method from affecting other applications in an application server, such as a Servlet platform, we can use program rewriting in the application server middleware. With a utility class `ThreadLocalProperties` which stores property key-value pairs in a thread-local map, we can rewrite the original `setProperty()` invocation code as:

```
ThreadLocalProperties.setProperty(
    "java.xml.sax.driver",
    "org.apache.xerces.parsers.SAXParser"))
```

Now the code invokes the method `setProperty()` on the utility class. Since a map stored in each thread-local object holds the property key-value pairs, the application server can control the locality of properties by maintaining the maps. By giving a new map to the thread-local object when a thread for application execution starts, the server can protect the system-wide properties from being changed through execution of an application.

2.2 Typical Implementation by Bytecode Rewriting

We can implement this kind of program rewriting either at load-time[9, 7, 4, 5, 12, 3] or at compile-time[14]. Figure 1 shows the typical flow of bytecode rewriting at load time. An application program or the bootstrap launcher for this transformation container triggers the loading of a class when they require the class in order to continue their execution. Loading requests are caught by one of the class loaders. When asked to load a class for the first time, a class loader extended for program transformation fetches the bytecode of the class from the original bytecode repository, such as a file or network. Instead of simply loading the fetched bytecode into a Java virtual machine (JVM), the extended loader lets a meta-program modify the bytecode before loading it

```
class Replacement extends ExprEditor {
    void edit(MethodCall m) {
        if (! m.getClassName().equals("java.lang.System"))  return;
        if (m.getMethodName().equals("setProperty"))
            m.replace("$_=ThreadLocalProperties.setProperty($$)");
        else if (m.getMethodName().equals("getProperty"))
            m.replace("$_=ThreadLocalProperties.getProperty($$)");
    }
}
```

Figure 2: Example rewriting code with Javassist.

into the JVM. It returns a class object for the loaded class, which can be obtained through the `defineClass()` method provided by the JVM.

A meta-program for this example seeks a method call expression in bytecode for every class. It replaces each method call with the `set/getProperty()` of `ThreadLocalProperties` if the method call meets these criteria:

- the target class of the method call is `java.lang.System`, and
- the method name called is `"getProperty"` or `"setProperty"`.

For example, using Javassist[3] this translation can be written as shown in Figure 2. The `MethodCall` instance represents a method call expression in the loaded bytecode. In this Javassist code, we replace the bytecode fragments of `System.set/getProperty()` with method calls on `ThreadLocalProperties` using `m.replace()`, which is part of the bytecode rewriting API provided by Javassist.

2.3 Problems with Transformations in Application Programs

Even though the solution by rewriting shown in Section 2.2 is typical and effective, it has a problem if application programs use the customizable class loader mechanism in Java.

Direct accesses to a system class loader

Direct invocation of class-loaders like system class loaders (the bootstrap class loader) also makes our `setProperty()` invocation on the `System` class escape from the net of code rewriting. For example, application programmers can write the following code in an application program to load a class. It avoids the rewriting process described above for the class `XMLLib`.

```
1: ClassLoader cl = ClassLoader.SystemClassLoader();
2: Class c = cl.load("app.XMLLibImpl");
3: XMLLib xmltool = c.newInstance();
4: xmltool.run();
```

This code first obtains the class loader object for the system class loader in line 1 and loads the class `app.XMLLibImpl` using that system class loader in line 2. It creates a new `XMLLibImpl` instance in line 3. Using the type `XMLLib`, in line 4, it invokes a method `run()` on `XMLLibImpl`. The `XMLLibImpl` may be written as follows to set a system property in it.

```
class AppClassLoader extends ClassLoader {
    public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
    }
    byte[] loadClassData(String name) {
        // load the class data from a specified location
    }
}
```

Figure 3: A user-defined class loader. This class loader fetches bytecode from the specified location and generates a class object using `defineClass()`.

```
class XMLLibImpl implements XMLLib {
  static void run() {
    System.setProperty(
        "java.xml.sax.driver",
        "org.apache.xerces.parsers.SAXParser");
    ....
```

The code replacement of `setProperty()` relies on the load-time program analysis and modification by a customized class loader. Since the class `XMLLibImpl` is not loaded by the rewriting class loader of the middleware, it would be activated as is without replacement of the `System.getProperty()`. Invoking `run()` on the loaded `XMLLibImpl` results in an undesired `setProperty()` invocation.

User-defined class loaders

Similarly to the use of a system class loader, the use of user-defined class loaders causes our `setProperty()` invocation on the `System` class to fail to be rewritten. For example, application programmers can define a class loader in an application program as shown in Figure 3 for controlling their own class loading behavior. Again, this avoids the rewriting process if classes are loaded through this user-defined class loader using code such as:

```
1: ClassLoader cl = new AppClassLoader();
2: Class c = cl.load("app.XMLLibImpl");
3: XMLLib xmltool = c.newInstance();
4: xmltool.run();
```

Figure 3 depicts bytecode in the scope of a customized class loader that analyzes and rewrites programs. The range of bytecode that a class loader can rewrite is only that which is loaded through that class loader. Classes loaded using a `load()` method on a class loader which is not a child of the rewriting class loader are inaccessible to the rewriting class loader. Even if classes are loaded using the `load()` method on a child class loader of the rewriting class loader, their loading is generally not delegated to the parent. They are also not rewritable in such cases. In addition, system (bootstrap) classes are prohibited to load by a customized class loader so their loading must be delegated to the parent system class loader, making it impossible to rewrite those classes.

3. EMPL - A FRAMEWORK FOR CAPTURING REFLECTIVE BEHAVIORS

We propose a framework called EMPL, attaining a comprehensive net for capturing reflective behaviors. This framework:

- allows program analyzing/rewriting software to detect and replace indirect method invocations which are not written in a visible way in application programs, and
- makes analysis and rewriting by middleware mandatory and transparent for modified application programs so that the transformation by the middleware is always certain to be performed on application programs.

Instead of requiring perfect, precise and static analysis of a reflective program, which is generally not possible, the proposed framework gives a comprehensive "net" for capturing the use of reflective computation in application programs at runtime.

A program in a meta-level layer (within analyzing/rewriting software) can completely control or sense the behavior and structure of programs in base-level layers overlaying the meta-level. This section presents the design and implementation of EMPL. The core of EMPL is a toolkit called the EMPL toolkit that performs transformation of bytecode so that proper use of meta-programming layers can be enforced in the transformed bytecode. First, we show the application programming interface for the EMPL toolkit and then explain how we implement the transformations.

3.1 The Application Programming Interface

The EMPL toolkit.

The EMPL toolkit provides a method:

```
Bytecode ensureEMPL(String name, byte[] buf,
                                 int offset, int length)
```

This method receives bytecode in a byte array and returns modified bytecode in the `Bytecode` object. The EMPL toolkit transforms the given bytecode so that meta-programming layers are certain to be applied there. The class `Bytecode` is just the holder of a class name, a byte array, and the offset and length of the bytecode stored in the array.

This method can be used in an analyzing/rewriting class loader. If a normal class loader code without EMPL appears as:

```
public Class findClass(String name) {
    byte[] b = ..;
    return defineClass(name, b, 0, b.length);
}
```

then a corresponding class loader which a programmer using EMPL should write is formed as follows:

```
public Class findClass(String name) {
    byte[] b = ..;
    Bytecode empl_b
        = EMPLToolkit.ensureEMPL(name, b, 0, b.length);
    return defineClass(empl_b.name, empl_b.buf,
                       empl_b.offset, empl_b.length);
}
```

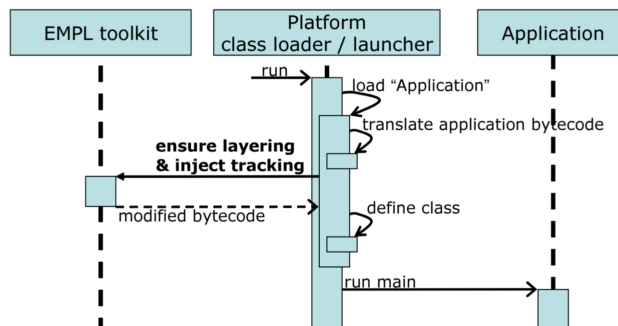

Figure 4: A sequence diagram for bytecode rewriting in a transformation container using EMPL. Before defining a class, the EMPL toolkit is invoked to transform the bytecode of the class to ensure access for the meta-programming layers there.

Figure 4 shows a sequence diagram for a transformation container using EMPL. Compared to the sequence diagram in Figure 1, a control flow edge goes to the EMPL toolkit after translating the bytecode of a class and before defining the class in JVM.

Configuring EMPL.

The launcher (start-up program) of a transformation platform (container) must configure its meta-programming layer using a method in the EMPL toolkit:

```
setLayer(ClassLoader loader,
         BytecodeTranforming translator)
```

The first parameter, `loader`, is a platform class loader in which the system performs program transformations. The second parameter, `reflTrap`, is a platform specific callback handler for trapping reflective member accesses in application programs running on the platform. The third parameter, `translator`, is also a callback handler, but for transforming the bytecode before it is loaded by the application-defined class loaders.

Typical configuration code in a platform launcher could be written as follows:

```
1: EMPLToolkit empl = new EMPLToolkit();
2: ClassLoader loader = new PlatformClassLoader(empl);
3: PlatformTracker tracker = new PlatformTracker();
4: empl.setLayer(loader, tracker);
5: Class appClass = loader.load(applicationClassName);
6: Method mainMethod = appClass.getMethod("main", ..);
7: mainMethod.invoke(null, ..);
```

```
public interface TranslationEnforcing {
    Bytecode enforceTranslation(String name,
        byte[] buf, int offset, int length) ..;
}
```

Figure 5: The interface `TranslationEnforcing` in the EMPL API. A callback handler for transforming bytecode loaded in application-defined class loaders must implement this interface.

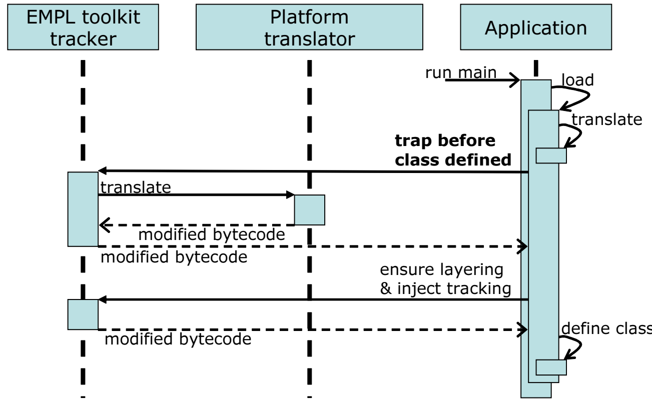

Figure 6: A sequence diagram for enforced program transformation in a user-defined class loader.

Callback for class loading.
A middleware programmer writes a callback handler for transforming bytecode loaded in the application-defined class loaders by implementing the interface `TranslationEnforcing` in Figure 5. Figure 7 shows how a handler is invoked by the EMPL system. Before program execution reaches the application code performing a class definition from bytecode in an application-defined class loader, the EMPL tracker traps the execution and invokes a registered callback handler offered by the platform middleware. After bytecode rewriting by the platform translator is done, the system transforms the resulting bytecode using the EMPL toolkit so that the bytecode conforms to the meta-programing layers.

3.2 Implementation

We implement the EMPL toolkit by bytecode rewriting. In the method `ensureEMPL()`, the toolkit seeks code which:

- retrieves a class loader through certain system classes, or
- defines a custom class loader

in bytecode passed as a parameter and inserts code or replaces their code so that the meta-programming layers are enforced there.

Transforming class loader accesses.
The objective of the first transformation is to avoid application classes being directly loaded by a system class loader.

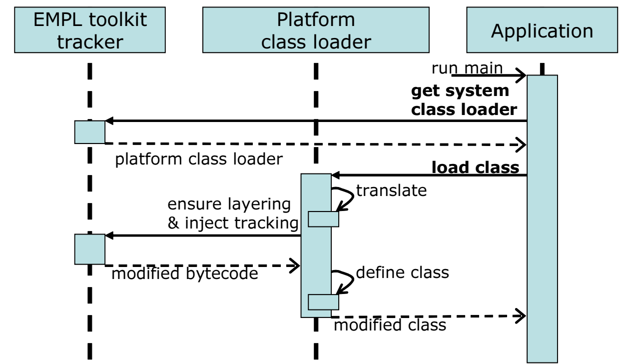

Figure 7: Trapped accesses to a system class loader. Any attempt to retrieve a system class loader is trapped by the EMPL system and a registered platform class loader is passed instead. As a result, the original code loading classes through a system class loader becomes code loading classes through the platform class loader.

The approach we take [1] for achieving this is:

- not to leak any reference to system loaders to application programs, and
- not to allow invocations on system class methods which load classes using a system class loader in their code.

For preventing leaks of system loader references, we transforms uses of the system methods that return `ClassLoader` objects. If a returned object is a reference to a system class loader, EMPL returns a registered platform class loader instead. General prevention of reference leaks requires more [16], but this transformation is sufficient in this case. Figure 7 shows a sequence diagram for execution with this transformation. As a result of the transformation, original code loading classes through a system class loader becomes code loading classes through the platform class loader.

As of the Java 2 Standard Edition under the version 1.4.x, the classes for which the uses are transformed by EMPL are: `Class.getClassLoader()`, `ClassLoader.getParent()`, `ClassLoader.getSystemClassLoader()`, `SecurityManager-.currentClassLoader()` and `Thread.getContextClassLoader()` in the `java.lang` package, and a few methods in other packages.

For example, the following code:

```
ClassLoader.getSystemClassLoader()
```

[1]There is an alternative transformation achieving the same effect of this transformation. In the alternative transformation approach, we can allow leaks of references to system loaders. Instead, we need to trap additional system methods; the EMPL toolkit must transform invocations on methods taking a class loader as a parameter and loading classes using the given class loader.

is transformed into the following code:

```
Tracker0.filterLoader(
    ClassLoader.getSystemClassLoader())
```

where the class Tracker0 is a class created at runtime by the EMPL toolkit for bridging between a toolkit instance and the base-level application programs. The method filterLoader() returns the registered platform class loader if the class loader given as a parameter is a system class loader.

In addition to the leak prevention of system class loader references, we need to take care of a few system methods which internally use a system class loader to load application classes whose names are given as parameters of the methods. For example, the method findSystemClass() of the class java.lang.ClassLoader allows a system class loader to load a class whose name is given to an invocation of this method as a parameter. EMPL replaces method calls with code loading classes through a registered platform class loader.

Moreover, it takes care of reflective access to class loaders in application programs. Fortunately in Java, classes providing reflective accesses to a class loader are limited to the Method class in the java.lang.reflect package. For method invocations, we only need to trap the declared reflective access method invoke() since, they are first (without overriding) declared in the final classes. It replaces every method calls through the invoke() with method calls to EMPL framework which examines whether method calls should be trapped by EMPL or not and traps them with EMPL handlers in the case.

Transforming user-defined class loaders.
The objective of the second transformation is to inject required transformation code into application-defined class loaders.

The code defining a class using the final method defineClass():

```
defineClass(name, buf, offset, length);
```

is transformed into:

```
Bytecode b1
    = Tracker0.forceTranslation(name, buf, offset,
                                length);
Bytecode b2
    = Tracker0.ensureEMPL(name, b1, 0, b1.length);
defineClass(b2.name, b2.buf, b2.offset, b2.length);
```

where the class Tracker0 is a class created at runtime by the EMPL toolkit for bridging between a toolkit instance and the base-level application programs.

Additionally, the superclass declarations of user-defined class loaders whose direct superclass was originally java.lang.ClassLoader are replaced with ones for the class EMPLLoader, which is a subclass of the class ClassLoader. In the class EMPLLoader, methods using defineClass() are overridden to invoke forceTranslation() and ensureEMPL() before the calls to defineClass().

Limitations.
System classes such as java.lang.String must be loaded by a system class loader thus they are not modifiable for EMPL. Transformations specified by users of EMPL are never applied to bytecode of system classes. Also this EMPL implementation by bytecode rewriting cannot capture reflective-/non-reflective accesses from native methods, where the accessing bytecode is not available.

3.3 The EMPL framework

With the EMPL framework, middleware programmers can reuse existing transformation containers without modifying their code by hand. To do so, they must provide a platform launcher wrapping for the original transformation container in addition to a callback handler for enforced program transformations.

For an application platform which can be launched as follows from a command line:

```
java platform.Launcher ...
```

programmers can write harness code for EMPL-ifying a launcher class of the middleware, platform.Launcher, as follows:

```
1: EMPLified platform
       = EMPLFramework.harness("platform.Launcher");
2: platform.setLoaderName("platform.Loader");
3: PlatformTracker tracker = new PlatformTracker();
4: platform.setLayer(tracker, tracker);
5: platform.run(args);
```

The EMPL framework transforms the class loader of the platform middleware so that it invokes the method ensureEMPL() of the EMPL toolkit before invoking defineClass(). A sequence diagram for the behavior of the EMPL framework launcher appears in Figure 8.

4. APPLICATION

EMPL is potentially useful for many middleware applications to enhance extension effects or enable them to function without causing unpredictable results. In this section, we first describe how EMPL can be used to solve the problem described in the example scenario in Section 2.3. After that, we mention other possible applications briefly.

4.1 Application to the Thread-Local Properties Example

We applied EMPL to the example scenario described in Section 2. In addition to the rewriting middleware partially shown in Figure 2, we needed to provide two EMPL callback handlers.

The utility class used in the transformed application code, ThreadLocalProperties, can be formed as shown in Figure 9. This class uses the class java.lang.ThreadLocal

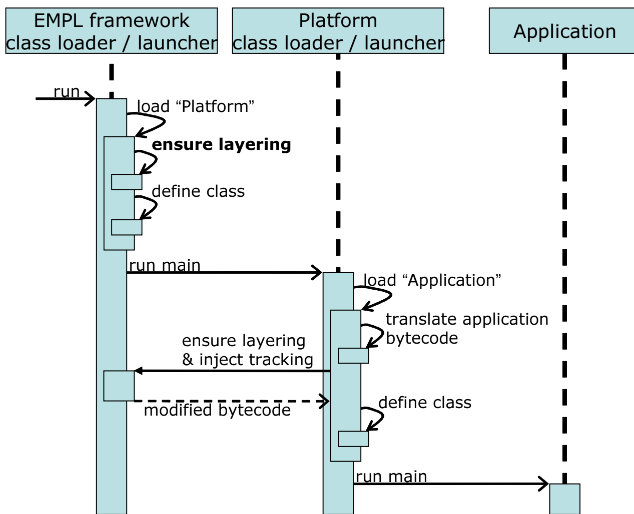

Figure 8: A sequence diagram for the EMPL framework launcher. It transforms the class loader of the platform middleware so that it invokes the method `ensureEMPL()` before invoking `defineClass()`.

```
class ThreadLocalProperties {
    static ThreadLocal tlv = new ThreadLocal();
    static String setProperty(String key, String value) {
        Map properties = (Map) tlv.get();
        return (String) properties.put(key, value);
    }
    static String getProperty(String key) {
        Map properties = (Map) tlv.get();
        String result = (String) properties.get(key);
        return (result != null) ? result
                                : System.getProperty(key);
    }
}
```

Figure 9: The class ThreadLocalProperties. This class is a substitute for `setProperty()` and `getProperty()` in the class `java.lang.System`. It maintains a thread-local property map.

provided in the standard Java API. All the properties set through the class are stored in a map held as a thread-local object. When `getProperty()` is invoked, the class first looks up the given key in the thread-local map and returns the corresponding value if it is found. Otherwise it delegates the request to `getProperty()` of the class `java.lang.System`.

We provide a callback handler for enforcing our transformation in the application-defined class loaders. If we use Javassist for bytecode editing, the code can be written as in Figure 10. The class `Replacement` used to transform each method body in the code is the same class as already shown in Figure 2 in Section 2.

4.2 Runtime overhead measurement.

The normal execution performance of application programs is not degraded by EMPL since EMPL inserts interception only in the class loader access code and reflective code. However, EMPL imposes a runtime overhead on reflective invocations and a load-time overhead for program transforma-

```
Bytecode enforceTranslation(String name, byte[] buf,
                            int offset, int length) {
    ClassPool cp = ClassPool.getDefault();
    cp.insertClassPath(new ByteArrayClassPath(name, .. buf ..));
    CtClass cc = cp.get(name);
    CtMethod[] methods = cc.getDeclaredMethods();
    for (int i = 0; i < methods.length; ++i)
        methods[i].instrument(new Replacement());
}
```

Figure 10: The program transformation enforcement for thread local properties. The class `Replacement` used here is the same class presented in Figure 2.

tions. We measured these overheads through experiments. Following experiments were run on the Sun Java 2 Runtime Environment, Standard Edition, version 1.4.2 (HotSpot Server VM) / Linux 2.4.18-14 / IBM IntelliStation M Pro with a single Pentium(R) 4 Processor 2.4GHz and 2GB of memory.

We measured the overhead of program transformations applied to application software available in the SPEC JVM98 benchmarks [2]. We prepared 2 platform middleware: middleware doing nothing (*empty*) and thread-local properties middleware (*tl*). We run SPEC's applications with the normal launchers and the EMPLified launchers (*XX-EMPL*). In addition to the 4 (2x2) types of application execution, we measured normal execution time of application (*normal*). We focused on the program startup regime of benchmark program execution by setting JVM option not to use any JIT compiler (-int option) and giving size 1 inputs to SPEC applications (-s1 option). (Figure 11.) Also, for seeing a steady state, we measured by setting JVM option to use a JIT compiler (-server option) and giving size 100 inputs to SPEC applications (-s100 option). (Figure 12.) We run the application benchmarks 3 times and took an average for each.

The results show slight overhead by EMPL. In addition to the overhead in class loading and reflective invocations, EMPL seems to affect the execution performance of normal code. This is because EMPL can interfere with some JIT compilation since it increases the code size of some method bodies. When there are relatively a lot of methods not executed frequently, the interference of JIT compilation results in a good score in a benchmark applications, and vice versa.

4.3 Possible Applications

Our experiences rewriting programs for distributed execution [15] are the original motivation of the work presented in this paper. In partitioned programs running on several hosts, which were originally a single program developed to run on a single host, remote proxy classes substitute for classes that exist on remote hosts. For example, a class `remote.File` is substituted for the class `java.io.File` on a GUI host, so that the file accesses on the host are interpreted as remote file accesses on a file system host. However, if there are application-defined class loaders, the program on a GUI host may try to access files that cause errors on the GUI host where the file system is not available. With

[2]http://www.spec.org/jvm98/

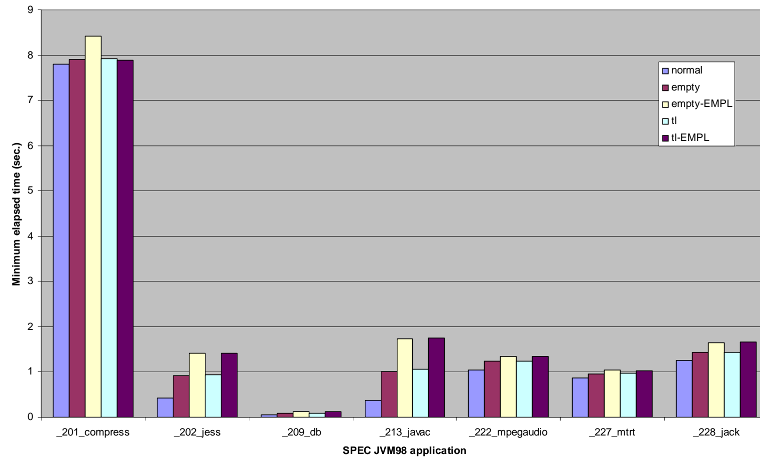

Figure 11: Elapsed times for overall application execution (interpreter, input size=1).

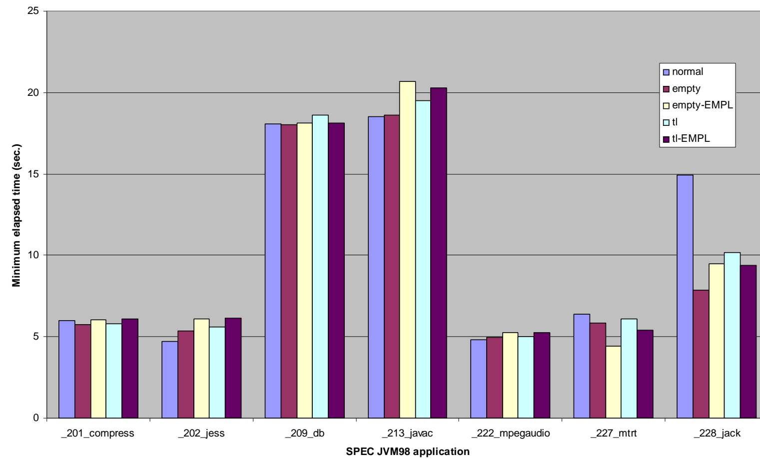

Figure 12: Elapsed times for overall application execution (JIT compiler, input size=100).

EMPL, we can avoid this situation.

Automatic replicated state management [2] requires notifying replicas of any state change in the replica's original source object. Field accesses on an original object must be changed so that they are done through getter/setter methods, which include the notification code in them. It relys on the code transformation for field access code so a field value can be directly changed without any notification if a customized class loader is used to load the accessor code. This results in an inconsistent state of the replicas. Using the EMPL toolkit, we can enforce rewriting of any field access code.

The `call` pointcut in AspectJ [8] allows advice code to be executed at the caller side of the designated classes. Implementation of an aspect weaver by compilation or bytecode rewriting rewrites the caller side of classes designated by a pointcut so that the corresponding advice code is inlined there or invoked as a method before, after or around the join-point. Use of customized class loaders easily bypasses this weaving implementation. The proposed framework can help the runtime system of AspectJ so that the system can always insert the advice code when specified methods are called.

Agesen et al. [1] have implemented a type-parameterizing extension of Java as an extended class loader. The extended loader accepts bytecode in their extended class file format for instantiating the parameterized classes at loading time. They reported that they needed to modify the JVM to avoid their implementation being too restrictive for general use. This is because bytecode in the extended format may be loaded when not desired by the system class loader if the application programs run on a regular JVM. The proposed framework can ensure that application bytecode should always be transformed before it is activated on a JVM.

Behavioral reflection [17] is often implemented using bytecode rewriting. In behavioral reflection, a meta-object intercepts field accesses on objects so that it can change the behavior of the objects of interest. Its implementation relies on rewrites of field access code. If an application program uses its own class loader in the program, this implementation does not work. The proposed framework can help behavioral reflection systems so that meta-objects can properly intercept any field access to objects even in this case.

5. CONCLUSION

The customizable class loader mechanism in Java provides the language with great flexibility but its use often imposes some restrictions on program transformation technologies. This paper addresses this problem. We are prototyping a framework called EMPL, which:

- makes program transformations by middleware mandatory and transparent for the modified application programs so that the transformation by the middleware is always ensured to be performed on the application programs, and
- runs on the regular JVM without modifying its implementation or its bootstrap class library.

The described design and implementation of EMPL employs a customized class loader on the regular Java virtual machine. Thus it is portable in the same way most of the transformation containers are. Moreover, since the system inserts interception only into reflective access code, the normal execution performance of application programs is not degraded by EMPL.

The wide applicability of EMPL in analyzing and rewriting middleware includes applications of automatic program distribution, replicated state management, aspect-oriented programming runtime systems, parameterized type extensions, behavioral reflection, and so on.

6. REFERENCES

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, number 10 in SIGPLAN Notices vol.32, pages 49–65. ACM, ACM Press, October 1997.

[2] G. C. Berry, J. S. Chase, G. A. Cohen, L. P. Cox, and A. Vahdat. Toward automatic state management for dynamic web services. In *Proceedings of the Network Storage Symposium (NetStore '99)*, Seattle, WA, USA, October 1999.

[3] S. Chiba. Load-time structural reflection in Java. In *ECOOP 2000 - Object Oriented Programming*, LNCS 1850, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[4] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with JOIE. In *USENIX Annual Technical Conference '98*, New Orleans, Louisiana, USA, June 1998. USENIX.

[5] M. Dahm. Byte code engineering with the javaclass api. Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, Berlin, Germany, July 1999.

[6] J. Gosling, B. Joy, and G. L. Steele Jr. *The Java Language Specification.* Addison-Wesley, 1997.

[7] R. Keller and U. Hölzle. Binary component adaptation. In E. Jul, editor, *ECOOP '98 — Object Oriented Programming*, LNCS 1445, pages 307–329, Brussels, Belgium, jul 1998. Springer-Verlag. July 20-24, 1998.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Grisworld. An overview of AspectJ. In L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[9] G. N. C. Kierby, R. Morrison, and D. W. Stemple. Linguistic reflection in Java. *Software — Practice and Experience*, 28(10):1045–1077, August 1998.

[10] G. Kniesel, P. Costanza, and M. Austermann". JMangler - a framework for load-time transformation of java class files. In *Proc. of IEEE Workshop on Source Code Analysis and Manipulation.* IEEE, 2001. none.

[11] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[12] C. Laffra, D. Lorch, D. Streeter, F. Tip, and J. Field. Jikes bytecode toolkit. Online, March 2000.

http://www.alphaworks.ibm.com/tech/jikesbt/.
[13] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98*, number 10 in SIGPLAN Notices vol.33, pages 36–44. ACM, ACM Press, 1998.
[14] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag, July 2000.
[15] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. In L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, LNCS 2072, pages 236–255, Budapest, Hungary, June 2001. Springer-Verlag.
[16] E. Tilevich and Y. Smaragdakis. J-Ohrchestra: Automatic Java application partitioning. In B. Magnusson, editor, *ECOOP 2002 - Object Oriented Programming*, LNCS 2374, pages 178–204, Malaga, Spain, June 2002. Springer-Verlag.
[17] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, LNCS 1616, pages 2–21, Saint-Malo, France, July 1999. Springer.