Instrumentation of Standard Libraries in Object-Oriented Languages: the Twin Class Hierarchy Approach

Michael Factor
IBM Research Lab in Haifa
Haifa University Campus
Haifa 31905, Israel
factor@il.ibm.com

Assaf Schuster
Israel Institute of Technology
Technion City
Haifa 32000, Israel
assaf@cs.technion.ac.il

Konstantin Shagin
Israel Institute of Technology
Technion City
Haifa 32000, Israel
konst@cs.technion.ac.il

ABSTRACT

Code instrumentation is widely used for a range of purposes that include profiling, debugging, visualization, logging, and distributed computing. Due to their special status within the language infrastructure, the *standard class libraries*, also known as *system classes,* provided by most contemporary object-oriented languages are difficult and sometimes impossible to instrument. If instrumented, the use of their rewritten versions within the instrumentation code is usually unavoidable. However, this is equivalent to 'instrumenting the instrumentation', and thus may lead to erroneous results. Consequently, most systems avoid rewriting system classes. We present a novel instrumentation strategy that alleviates the above problems by renaming the instrumented classes. The proposed approach does not require any modifications to the language, compiler or runtime. It allows system classes to be instrumented both statically and dynamically. In fact, this is the first technique that enables dynamic instrumentation of Java system classes without modification of any runtime components. We demonstrate our approach by implementing two instrumentation-based systems: a memory profiler and a distributed runtime for Java.

Categories and Subject Descriptors

D.1.5 [**PROGRAMMING TECHNIQUES**]: Object-oriented Programming; D.2.7 [**SOFTWARE ENGINEERING**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Languages

Keywords

Code Instrumentation, Standard Class Libraries, Inheritance, Java

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
Copyright 2004 ACM 1-58113-831-8/04/0010 ...$5.00.

1. INTRODUCTION

Code instrumentation is the act of modifying the code of an existing application by inserting new code statements and modifying or deleting existing ones. Instrumentation may be applied to source code as well as to compiled code. In the latter case, it may be applied statically, before the execution begins, or dynamically at run time.

Code instrumentation is a powerful mechanism for understanding and modifying program behavior. It is employed in various fields, including debugging, logging, visualization, access control, performance evaluation, distributed computing, and aspect-oriented programming. (Note that the latter can be used in any of the other fields.)

Many modern object-oriented languages, *e.g.*, Java, C#, Eiffel, Smalltalk, O'Caml and Objective-C, supply a rich set of reusable core classes, known as *standard class libraries* or *system classes*. These classes improve the usability of a language, allowing the programmer to concentrate on the higher-level tasks. A subset of standard classes provides convenient interfaces to operating system facilities, *e.g.*, I/O and networking. Henceforth, we will use the terms *system classes* and *user classes* to designate the standard library classes and user-defined classes, respectively.

In general, instrumentation may affect any code used by the original program. In the context of object-oriented languages, the instrumentation process may modify system classes. If the inserted code utilizes the instrumented system classes, it may lead to incorrect results. This is because the functionality of the instrumented system classes is different from the original. To put it simply, using instrumented classes in the inserted code is equivalent to 'instrumenting the instrumentation'.

For instance, consider a profiler for Java that instruments Java *bytecode* to record the sizes of objects created by the application. After each object creation statement, it inserts code that stores the size of the new object in an instance of a system class `java.util.LinkedList` by calling its `add` method. Assume that the original application also uses this list class. Therefore, during profiling, the application must use the instrumented list class to detect the objects created in its own instances of the list class. However, if the code added by the profiler uses the instrumented version as well, infinite recursion occurs. When an object is created, the profiler invokes the `add` method of the list; this method creates an object representing a new list entry. If the list class used in the instrumentation code is itself instrumented, the creation of a new entry in the `add` method will be followed

by another invocation of `add`, leading to infinite recursion. Even without the recursion problem, the profiler would yield incorrect results, recording the sizes of the list entry objects created by the instrumentation code.

To produce correct results, the instrumentation code must use the original system classes, while all other code uses their instrumented counterparts. (Certain types of instrumentation may still require that the instrumented system classes be used.) Therefore, the runtime must use the original and instrumented versions of the same class simultaneously. This, however, is problematic because both versions have the same name, *i.e.*, there is a *name clash problem*.

We propose an instrumentation approach that allows both the original and instrumented versions of a system class to coexist within the same execution environment. It solves the name clash problem by renaming the instrumented classes and modifying all code, other than the instrumentation code, to use these renamed classes. The inheritance hierarchy of the instrumented classes is isomorphic to the original one. Therefore, we call our solution the *Twin Class Hierarchy* approach (TCH).

One of the most important features of TCH is its portability. Being entirely based on instrumentation, it is orthogonal to the implementation of the targeted language framework. Neither the compiler nor the runtime environment need to be modified. Since in most cases the initial goal of code instrumentation is portability, a nonportable solution would be unacceptable.

In most languages, the TCH class name transformation does not preserve the integrity of the code and may come into conflict with features such as inheritance, exceptions, and reflection. Therefore, we augment the TCH approach with techniques that overcome these difficulties. Despite the language-specific nature of several problematic features, most of our solutions are generic, or at least can be employed by the popular contemporary languages, *e.g.*, C#, Smalltalk, and Java. The implementation of TCH is not entirely automatic, but may require a certain amount of hand tuning to adapt it to a particular language. However, once the tuning is completed, TCH can be used automatically by any general instrumentation in that language.

Of course, one could argue that the problem could be solved by avoiding the use of system classes in the inserted code. However, this restriction significantly limits the convenience and power of the instrumentation. Instead of reusing system classes, the user would have to reimplement them. In addition, the functionality of those classes that cannot be reimplemented would become unavailable. For example, in the Microsoft Intermediate Language (MSIL) used in .NET, it is impossible to synchronize thread activities without using classes from the `System.Threading` package.

Their special status within the runtime makes system classes difficult to instrument. In fact, most frameworks, ([1, 2, 4, 15, 16] for example), avoid instrumenting them, modifying only user classes. Some systems [1, 2, 4] limit their functionality to user classes, while others [15, 16] invest considerable effort into implementing context-specific workarounds.

TCH facilitates the instrumentation of system classes. With TCH class name transformation, the instrumented system classes become user classes, which are easily instrumented by most frameworks. In particular, as we show in Section 3, TCH allows Java system classes to be instrumented, statically or dynamically, without any modification of the Java Virtual Machine (JVM) or the operating system components. By contrast, existing frameworks that are capable of instrumenting Java system classes compromise portability. For example, Keller et al. [12] make changes in the JVM. Duncan et al. [10] modify the dynamically linked libraries (DLLs) that are used to access the file system.

Although our main motivation is to allow the original system classes to be used, our solution can be applied to user classes as well. This is useful when a reusable non-standard library, *e.g.*, a library for management of log files, employed by the application to be instrumented, is also utilized in the code inserted by the instrumentation. We focus on system classes because it is the more difficult problem.

We have employed the TCH approach in an instrumentation-based profiler for Java and in a distributed runtime for Java, which we call JavaSplit [11]. The profiler collects statistics about memory allocations performed by an application. We use it to explore the *SPECjbb* benchmark [5]. The JavaSplit runtime instruments a standard multithreaded Java program for distributed execution. The instrumentation intercepts events that are interesting in the context of distributed execution, *e.g.*, accesses to shared data, synchronization, and creation of new threads.

In both systems, the inserted code uses the original versions of system classes, while all other code uses their instrumented counterparts. The ability of TCH to instrument all system classes plays an important role in the implementation of both systems. Without it, the profiler would not be able to detect many of the allocations that occur within the system classes, whereas JavaSplit would not be able to maintain the correctness of memory and synchronization operations.

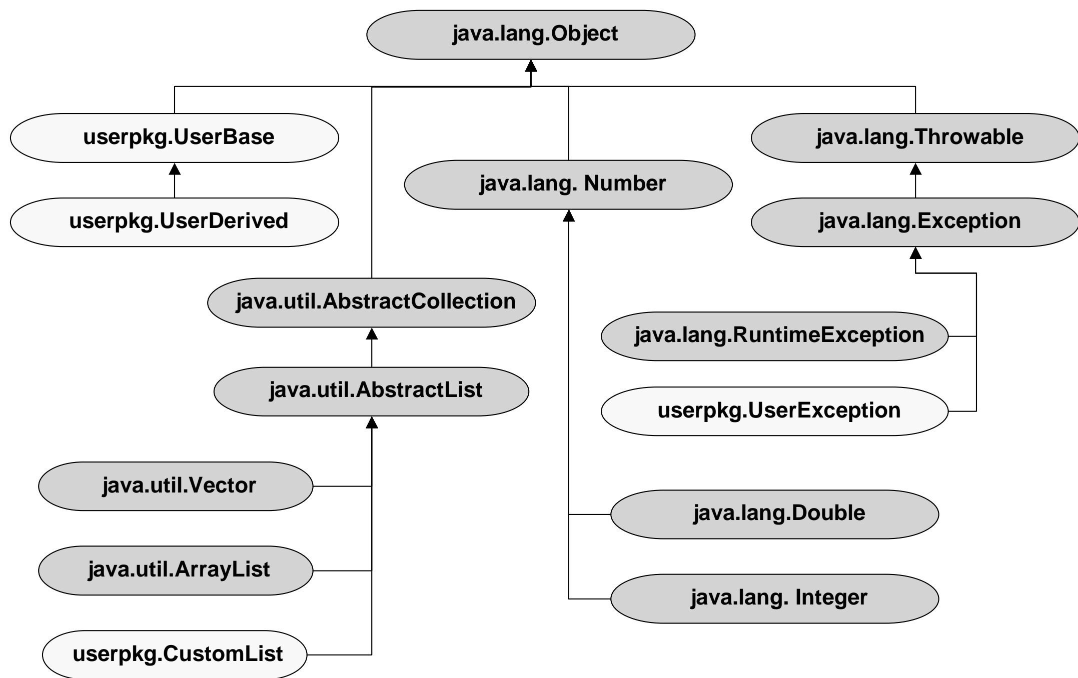
The main contributions of this paper are as follows. First, we present a novel instrumentation strategy that extends the capabilities of code instrumentation in object-oriented languages. Second, we discuss issues that arise when implementing our strategy in contemporary languages. Finally, we show how TCH enables dynamic instrumentation of Java system classes.

The structure of the rest of this paper is as follows. Section 2 presents the Twin Class Hierarchy approach. In Section 3 we discuss the difficulties of instrumenting system classes and show how TCH alleviates them. Section 4 explores the overhead of TCH. Section 5 demonstrates the contribution of TCH in profiling and distributed computing. In Section 6 we present approaches similar to TCH. Section 7 describes several load-time instrumentation frameworks and their relation to TCH. We conclude in Section 8.

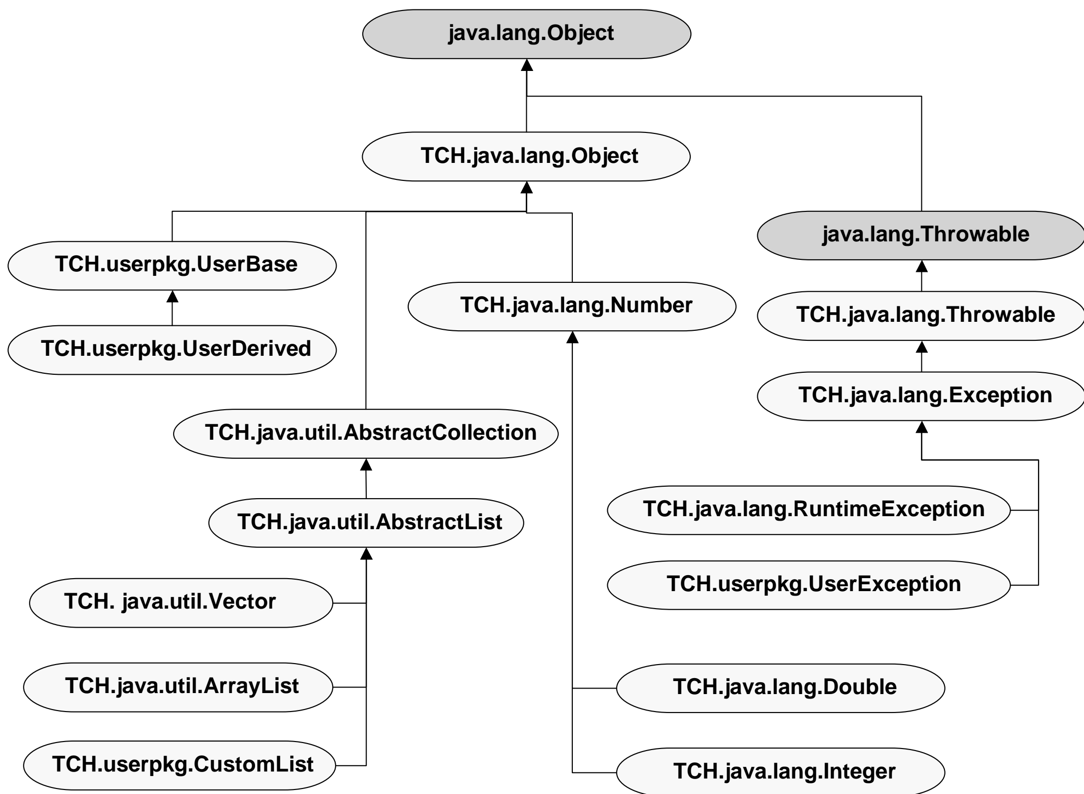
2. TWIN CLASS HIERARCHY

At the heart of the TCH approach lies the idea of renaming the instrumented classes to allow the instrumented and original classes to be used simultaneously by the runtime. For each original user or system class we produce an instrumented version with a different name. The inheritance relations of the instrumented classes mimic the original inheritance hierarchy. Thus, the new hierarchy is isomorphic to the original one. Figure 1 illustrates class renaming in Java. (The new name is produced by adding a prefix '`TCH.`' to the original one, so that `SomeClass` becomes `TCH.SomeClass`.)

In the instrumented version of a class, all code other than that inserted by the instrumentation is modified to use



(a) Original class hierarchy



(b) Twin class hierarchy

Figure 1: A fragment of the class hierarchy in Java, before and after the TCH transformation. The instrumented versions of system classes become user classes. (Java system classes are designated by dark gray.) The irregularity of `java.lang.Throwable` is discussed in Section 2.3.

```java
public class Example
                extends java.util.Vector
{
  private int intField;

  protected java.lang.Long longObjfield;

  private java.lang.Object objField;

  // constructor
  public Example{
    ...
  }

  java.util.List someMethod(
        int n, java.lang.String s) {
    ...
  }

  java.lang.Integer someOtherMethod (
        java.util.Vector vec,
        java.lang.Object obj)
  {
    ...
    if(obj instanceof java.lang.String){
        ...
    }
    ..
    java.lang.Integer local
              = new java.lang.Integer(n);
    java.lang.System.out.println(local);
    ...
    return localVar;
  }
}
```

(a) Original class

```java
public class TCH.Example
        extends TCH.java.util.Vector
{
  private int intField;

  protected TCH.java.lang.Long longObjfield;

  private java.lang.Object objField;

  // constructor
  public Example{
    ...
  }

  TCH.java.util.List someMethod(
    int n, TCH.java.lang.String s) {
    ...
  }

  TCH.java.lang.Integer someOtherMethod(
        TCH.java.util.Vector vec,
        java.lang.Object obj)
  {
    ...
    if(obj instanceof TCH.java.lang.String){
        ...
    }
    ..
    TCH.java.lang.Integer local
              = new TCH.java.lang.Integer(n);
    TCH.java.lang.System.out.println(local);
    ...
    return localVar;
  }
}
```

(b) Twin class

Figure 2: A Java class before and after the TCH transformation. (Although we present source code, the actual transformation can be performed on bytecode as well.) All referenced class names, except `java.lang.Object`, are replaced.

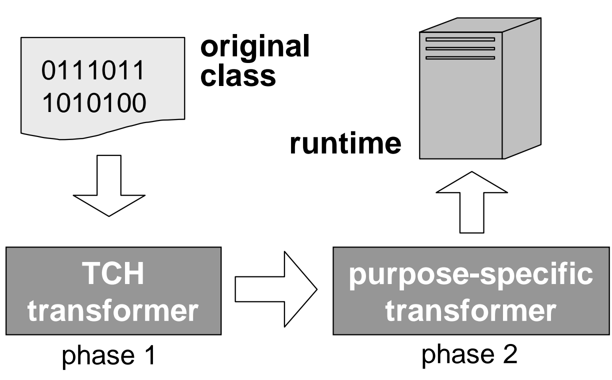


Figure 3: Instrumentation process. The code inserted during the second phase uses original class names.

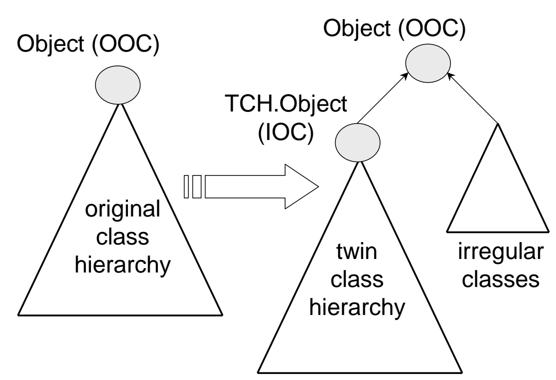


Figure 4: High-level view of class hierarchy transformation

the TCH class names (see Figure 2). For example, in Java bytecode, the renaming affects the instructions `instanceof`, `invokevirtual`, `new`, `getfield`, *etc.* (See [14] for the exact semantics of the above instructions.) However, as we show in Section 2.7, there is no need to modify the strings that designate class names. (In fact, they cannot always be distinguished from the other strings.)

The TCH-related transformations are independent of the purpose-specific instrumentation. Therefore, it is possible to rewrite the code in two phases, applying the TCH transformations prior to the purpose-specific transformations (see Figure 3). To solve the circular dependency problems described in the previous section, the latter phase inserts code that uses the original class names. Both phases can be performed statically, or both can be performed dynamically. It is also possible to produce the TCH versions of classes statically, and then apply the purpose-specific transformations at run time. The TCH phase is implemented only once per language. When implemented, it can be reused with any general instrumentation process.

The class name transformation, if performed naively, compromises the integrity of the instrumented code in several contexts, including inheritance, reflection, and exception handling. In this section we study the problems that emerge and present techniques to resolve them. Together with the modification of class names, the presented solutions constitute the previously mentioned TCH transformations. In this section, the term *instrumented class* means *class resulting from applying the TCH transformations on the original class.*

2.1 Root class

Most modern object-oriented languages, *e.g.*, Java, Smalltalk and C#, have an *object* class at the top of their class hierarchy (`java.lang.Object` in Java, `Object` in Smalltalk, `System.Object` in C#). Due to the name change, the *instrumented object class* (IOC) is not the top hierarchy class, but a subclass of the *original object class* (OOC).

Figure 4 presents a high-level view of the class hierarchy transformation. In the new class hierarchy, the IOC is a *direct* subclass of the OOC. Since the IOC is not the root class, the language may have certain special classes that are not its subclasses. For example, in C#, the TCH transformation cannot replace the superclass of an array class, because arrays are implicitly defined (using the `[ ]` operator at the source code level). As a result, despite the TCH transformation, arrays remain subclasses of `System.Object` and are not subclasses of `TCH.System.Object`. Classes that do not subclass the IOC are denoted *irregular classes*. These classes present a few problems that are discussed and resolved in Section 2.2.

2.2 Irregular classes

The presence of irregular classes is problematic mainly in languages with static typing, *e.g.*, Java, C# and O'Caml. Languages with dynamic typing (such as Smalltalk) may be affected by it, depending on whether the purpose-specific instrumentation modifies OOC behavior.

In statically typed languages, TCH class renaming transforms all uses of OOC in the original code, *e.g.*, local variables, class fields, and method parameters, into uses of IOC. Thus, a variable of type OOC that originally could have referenced an irregular class, *e.g.*, an array class, is transformed to a variable that is unable to do so, because irregular classes do not subclass IOC. For example, in instrumented C# code, a local variable of type `TCH.System.Object` cannot contain a reference to an array of integers whose direct superclass remains `System.Object`, despite the class name transformation. To solve this problem, OOC references are not replaced by IOC references in the instrumented code. Rather, they are left as they are, in order to allow them to reference the irregular classes (see Figure 2). The only exception to this rule is made in the code that originally creates an OOC instance: after instrumentation it creates an IOC instance.

```
class TCH.IntArrayWrapper
extends TCH.Object
{
  private int[] array_;
  // constructor, substitutes NEWARRAY
  public TCH.IntArrayWrapper(int size){
    array_ = new int[size];
  }
  // substitutes IALOAD
  final public int load(int index){
    return array_[index];
  }
  // substitutes IASTORE
  final public void store(int index,
                          int value){
    array_[index] = value;
  }
}
```

Figure 5: A possible wrapper for a Java integer array. Accessor methods are `final` to enable inlining.

Another problem with irregular classes is that they are not affected by the changes made by a purpose-specific instrumentation in the implementation of IOC. For example, if a field `myID_` and a method `id()` returning this field are added to IOC, only the subclasses of IOC will have an ID. This problem, which is relevant also in languages with dynamic typing, may be resolved by applying the transformations applied to the IOC directly to the implementation of irregular classes. In the above example, all irregular classes should be augmented with the ID field and ID method. Unfortunately, this solution is inapplicable to classes that do not have a class definition, but rather are defined implicitly, *e.g.*, array classes. In these special cases, the problem is solved by defining a wrapper class around the implicitly defined classes. This wrapper should subclass the IOC, thus inheriting all its functionality. All references to the original class should be replaced with references to the wrapper. Figure 5 illustrates a wrapper for a one-dimensional Java integer array. In all the instrumented bytecode the invocations of the bytecode instructions `newarray`, `iaload` and `iastore` should be replaced by invocations of the wrapper constructor, and methods `load` and `store` respectively.

In statically typed languages, a variable of type OOC that points to a subclass of IOC needs to be downcast to IOC whenever accessing a method or a (public) field added to IOC by the purpose-specific instrumentation. If the object referenced by a variable of type OOC is an instance of an irregular class, then it needs to be downcast to its own class,

before accessing the additional member. (If the irregular class was not augmented with these members, the code accessing them must be skipped.)

2.3 Classes with special semantics

Some languages attribute special semantics to certain classes (or other class-like constructs, *e.g.*, Java *interfaces*). For instance, in Java, only a subclass of `java.lang.Throwable` can be thrown as an exception. Only a class implementing the `java.io.Serializable` interface can be marshaled into a bitstream. The renaming causes the instrumented classes to lose their special semantics, often violating the integrity of the instrumented code. For example, in the instrumented code the argument of a `throw` statement in Java is no longer a subclass of `java.lang.Throwable`, but rather a subclass of `TCH.java.lang.Throwable`, which is illegal.

Since the special semantics are class-specific, so in theory should be the solution. In practice, a general technique presented below solves the problem for those cases of special class semantics of which we are aware. (Note that very few system classes have unusual semantics.)

In order to regain the special semantics of an instrumented class, we make it a direct subclass of its original version. This solves the problem, because it creates an *'is a'* relationship between the instrumented class and its original counterpart. For instance, in the case of Java's throwable class, the instrumented class `TCH.java.lang.Throwable` becomes a subclass of `java.lang.Throwable` (Figure 1(b)) and therefore can be used in a `throw` statement. This solution also works for all of Java's special interfaces. When this technique is applied, additional irregular classes are created.

The above solution would not be possible unless the original special classes could be subclassed. For example, it would not work if Java's throwable class were a `final` class. (In Java, `final` classes cannot be subclassed.) It would also fail if the original class was not a direct subclass of the OOC. Fortunately, those classes with special semantics can be subclassed and directly subclass the OOC.

2.4 Illegal method overriding

In most languages with static typing, an inherited method cannot be overridden by a method with a different return type and the same argument types. In previous sections we saw cases of instrumented classes subclassing their original counterparts. In section 2.1, we describe how an IOC is made a direct subclass of an OOC. In section 2.3 we show how instrumented classes regain their special semantics by subclassing their original versions. One consequence of the TCH class name transformation is that an instrumented class may contain a method that differs from the original method only by a return type. (Method arguments remain the same if a method does not have object reference arguments.) Therefore, subclassing a non-TCH class by its instrumented version can lead to a violation of method overriding rules. For example, in Java, the return type of the method `toString()` in IOC is `TCH.java.lang.String`, which is different from its return type in OOC.

We solve this problem by renaming the problematic methods, and modifying the code to use the new names, which are produced by adding the prefix '`TCH__`' to the original names. For example, when creating the IOC, its `toString()` method is renamed `TCH__toString()`.

2.5 Object constants

Most languages contain object constants. For instance, in Java, string literals are objects of type `java.lang.String`. In pure object-oriented languages like Smalltalk, all constants and strings, *e.g.*, 1977, 0.333 and 'hi', are objects.

Since object constants are instances of the original system classes, they are not compatible with the rest of the code, which is instrumented to use the TCH classes. We solve this problem by replacing each object constant of a class `A` with an instance of `TCH.A`. For example, in Java, each string literal is replaced with a corresponding instance of `TCH.java.lang.String`. (A regular string may be converted into a TCH string by using the underlying character array of the former to create an instance of the latter.) The above problem exists both in statically and dynamically typed languages. In the latter, if object constants are not converted to the TCH form, they will not be affected by the instrumentation.

2.6 Runtime exceptions

Runtime environments, *e.g.*, JVM or .NET Common Language Runtime (CLR), may throw runtime exceptions. These exceptions are not TCH objects.

In statically typed languages the exceptions are caught on the basis of their class. The TCH class name transformation causes the rewritten catch statements to catch the TCH exceptions but miss the runtime exceptions. To solve this problem we modify each catch statement to catch both the instrumented and the original versions of the exception classes associated with it. In both dynamically and statically typed languages, a caught runtime exception is converted, before the execution of an exception handler, to an instance of the corresponding TCH class. This ensures that the exception handler will process a TCH version of the exception.

2.7 Reflection

The TCH approach modifies class names and method names (Section 2.4). Consequently, it must be adapted to preserve the behavior of the reflection mechanisms of a language. Since reflection may be implemented differently in each language, there is no generic solution for this problem. However, the following simple technique allows the support of the basic features of reflection in most languages.

In the instrumented code, when invoking reflection related methods, we perform translation between the original and the instrumented class (or method) names. Each input parameter of a reflection method designating the name of a class (or a method) is converted to the TCH form (by addition of the TCH prefix). Each output (usually the return type) designating the name of a class (or a method) is converted to the non-TCH form. (Note that if a name is already in the desired form, we leave it as is.) The translation is performed inside the reflection-related methods.

For example, consider the methods `forName` and `getName` of the Java class `java.lang.Class`. The former method returns an instance of `java.lang.Class` that corresponds to its single string parameter. The latter method, which does not have any parameters, returns the name of a class represented by the given instance of `java.lang.Class`. In the instrumented code, the class name parameter of `forName`, *'SomeClass'* is converted to *'TCH.SomeClass'*. Therefore, the returned instrumented class object represents the instrumented class rather than the original class. Thus, if

```
class TCH.File extends TCH.Object {
  private File origImpl_;
  // constructor
  public File(TCH.String name){
    origImpl_=
     new File(name.TCH__toOrigString());
  }
  // write a byte to a file
  public void write(byte b){
    origImpl_.write(b);
  }
  // read a byte from a file
  public byte read(){
    return origImpl_.read();
  }
  ... // other API methods
}
```

Figure 6: Implementation of an instrumented native class using the delegation pattern (in Java-like coding style). Note the string conversion in the constructor.

we use the returned class to create an instance, we will create an instance of the instrumented class and not the original. Similarly, the instrumented string returned by `getName`, *'TCH.SomeClass'* is translated to *'SomeClass'*. Consequently, if the returned string is compared to a hardcoded class name string, the result of the comparison will be correct because the hardcoded names remain in non-TCH form, despite the TCH transformations.

2.8 Built-in classes

Contemporary languages have system classes whose implementation is integrated into the runtime. More specifically, the implementation of a subset of their methods is hardcoded. Such methods are called *native* in Java and *internalcalls* in C#. Henceforth, we conform to Java terminology and call them *native methods*. System classes that contain native methods will be referred to as *native classes*.

Implementation of a native method is always bound to a particular method in a particular class. It can be accessed only by calling that method, but cannot be reused in the implementation of another method (or in another class). For example, consider the native method `currentTimeMillis()` defined in the Java class `java.lang.System`. Its implementation, which queries the underlying OS for the current time, cannot be incorporated into some other class. Therefore, after TCH class renaming, the native functionality is unavailable in the instrumented system classes. To correct this, TCH must provide the instrumented system classes with an alternative implementation that simulates the original API. We accomplish this by using the original version of a class in the implementation of its instrumented counterpart. This is possible only because the TCH approach allows the original and the instrumented versions of a class to coexist.

In most cases, an instrumented version of a native class is implemented as a wrapper around the original class. The wrapper methods delegate invocations of the native API methods to an encapsulated instance of the original class. Figure 6 illustrates the implementation of the instrumented version of a hypothetical class `File` that originally had native API methods to access the file it represents. If necessary, the wrapper methods convert the parameters and the return type from TCH form to the original form and vice-versa.

Normally, only a small portion of the system classes are native. Most of these are related to reflection, GUI, I/O, and networking. In Java they constitute about 3% of the system classes. A much smaller portion is required to run most programs that do not contain a GUI. We have successfully executed various applications, including SPECjbb, and applications that perform I/O and networking.

2.9 Applicability discussion

Some techniques used to alleviate the side effects of the TCH approach are language specific. Therefore, in theory there can be a language to which TCH would be inapplicable.

TCH is a general methodology and not an algorithm. Thus, the techniques to alleviate its side effects should be perceived as guidelines rather than specific instructions. If a certain issue cannot be resolved by the proposed techniques, then language-specific solutions should be sought. The strength of TCH is in the fact that once all issues are resolved for a particular language, it can be automatically employed by any general purpose instrumentation in that language.

3. FACILITATING INSTRUMENTATION OF SYSTEM CLASSES

Code instrumentation can be performed statically, before the execution begins, or dynamically at run time. The special status of system classes makes their transformation problematic in both modes. TCH eliminates most of the difficulties, because, after renaming, the instrumented versions of system classes are no longer part of the standard class libraries but are rather user classes, which are much easier to instrument. In contrast to the instrumentation frameworks that allow arbitrary transformations of all system classes, TCH does not require modification of any components of the language infrastructure, *e.g.*, the compiler or the runtime.

In this section, due to a variety of language-specific mechanisms and issues, we focus our discussion on Java. However, most of it is valid for other frameworks with dynamic loading, including the .NET platform.

3.1 Static instrumentation

If the instrumentation does not modify the class names, static instrumentation of system classes must force the runtime to use the instrumented versions instead of the originals. This may not always be possible, because the system classes may be deeply integrated into the runtime.

In Java, most popular JVMs, *e.g.*, Sun and IBM JDKs, provide a command line option that allows the user to specify the path from which system classes should be loaded. In the JDKs mentioned above, it is also possible to change the implementation of system classes by modifying the contents of the file `rt.jar` in which most of the system classes are stored. Unfortunately, neither option is standardized. Therefore, in theory, there may be a valid JVM that does not allow static instrumentation of system classes.

With TCH class name transformation, the instrumented system classes become user classes. Therefore, they are

loaded as ordinary user classes, not instead of, but in addition to their original versions, rendering the need to replace them or update their loading path obsolete. Thus, TCH avoids potential portability problems.

Runtime environments can make assumptions regarding the structure and loading order of system classes. If these assumptions do not hold, a runtime may terminate abnormally, often without a comprehensive error message. For example, most JVM implementations make assumptions about the size of the classes `java.lang.Object`, `java.lang.Class` and `java.lang.String`. If the instrumentation process augments one of these classes with a field, the JVM crashes. It will also crash if the loading order is changed as a result of instrumenting the above classes. Note that these problems arise not only in static but also in dynamic instrumentation. TCH lets the JVM keep the original definitions of the problematic classes, thus avoiding these difficulties.

3.2 Dynamic instrumentation

The main advantage of dynamic instrumentation is that it does not require *a priori* knowledge of the classes used by a program (*closed world assumption*). Since reflection allows the loading of classes whose identity is determined at run time, it may be impossible to determine the transitive closure of classes used by a program. Moreover, classes created at run time can only be instrumented dynamically.

The most important challenge in dynamic instrumentation is to intercept all the classes employed by an application. In most runtimes, it is difficult to intercept system classes. In Java, there are two main obstacles. Both are related to the Java class loading mechanism, which is used by most contemporary frameworks to implement dynamic instrumentation. First, a subset of system classes (approximately 200 in Sun JDK 1.4.2) is already loaded by the JVM, before the class loading mechanism can be modified to enable rewriting. Since most of these *preloaded* classes are used extensively by non-trivial applications, it is important that they can be instrumented. (Among the preloaded classes are: java.lang.Integer, `java.lang.String, java.util.HashMap, java.lang.Thread`.) Second, the class loading mechanism attempts to ensure that the system classes are defined by the *bootstrap class loader*. Since the bootstrap class loader is integrated into the JVM, the user cannot gain any control over it without modifying the JVM, which is highly undesirable. Consequently, it is hard to modify the definition of a system class.

Existing portable load-time instrumentation frameworks, such as JMangler [13], Javassist [7], JOIE [8], BCEL [9], AspectWerkz [2], and JBoss AOP [4], do not instrument system classes (due to the problems mentioned above). The frameworks that are capable of instrumenting system classes, *e.g.*, [12] and [10], compromise portability by modifying the JVM or the underlying DLLs. Our recent communication with the representatives of JMangler, AspectWerkz, and JBoss AOP has revealed their desire to perform dynamic transformation of system classes as well as their inability to accomplish this.

The TCH approach accomplishes dynamic instrumentation of Java system classes. TCH does not suffer from the problems mentioned above because it renames system classes, thus transforming them into user classes. Since most runtimes (with dynamic loading) allow dynamic instrumentation of user classes, TCH effectively allows all system classes to be instrumented without any modification of the runtime infrastructure. To the best of our knowledge, TCH is the only technique that achieves this in Java.

```
IF THE NAME OF REQUESTED CLASS STARTS
WITH 'TCH.':
    READ THE ORIGINAL CLASS
    PRODUCE ITS TCH VERSION
    APPLY THE PURPOSE-SPECIFIC TRANSFORMATIONS
OTHERWISE: LOAD THE CLASS IN THE USUAL WAY
```

Figure 7: Dynamic class loading procedure

3.2.1 TCH-based dynamic instrumentation in Java

In Java and similar frameworks, such as .NET Common Language Runtime, TCH supports dynamic instrumentation in the following way. Let `AppMain` be the class that contains the `main` method of the application to be executed. At the beginning of the execution, we convert the string parameters of the main method into TCH form. Then, we hook into the class loading system either by installing a custom class loader, as is done in Javassist, or by replacing the definition of the system class loader (`java.lang.ClassLoader`), as is done in JMangler. After that, we instruct the adapted class loading system to load `TCH.AppMain`, and then employ reflection to execute its main method.

When asked to load a class whose name begins with '`TCH.`', *e.g.*, `TCH.somePackage.SomeClass`, the adapted class loading mechanism fetches the definition of the corresponding original class (`somePackage.SomeClass`) from the loading path, and then sequentially applies to it the TCH and purpose-specific transformations, as described in the beginning of Section 2. When asked to load a class whose name does not begin with '`TCH.`', the adapted class loader loads the class without applying any transformations to it. The above procedure is summarized in Figure 7.

The entire twin hierarchy, including the system classes, is produced on the fly. The implementation of TCH versions of native system classes described in Section 2.8 is hardcoded into the transformer and thus can also be generated dynamically. Alternatively, the TCH versions of system classes can be produced statically, while the TCH versions of user classes are produced dynamically.

4. TCH OVERHEAD ANALYSIS

We estimate the overhead of TCH using sequential applications from the Java Grande Forum (JGF) Benchmark Suite (version 2.0) [6] and the SPECjbb benchmark [5], which is probably the most important existing benchmark for server-side Java. We compare the throughput of the original programs with their TCH counterparts, which are produced statically. The measurements were performed on Intel's dual-processor machine, 2x1.7 GHz with 1 GB memory, using Sun JDK 1.4.2.

The sequential benchmarks in the JGF benchmark suite are divided into three categories. The first one measures the performance of low level operations such as arithmetic, casts, assignments, allocation of data, exceptions, loops, and method invocations. The applications in the second cate-

Table 1: JGF Benchmark Suite – microbenchmark results. Due to space limitations we present only a subset of benchmarks. The throughput difference of the omitted benchmarks is insignificant.

Benchmark	Original	TCH	Difference (%)	Units
Arith:Add:Int	864733630	852847490	1.37	(adds/s)
Arith:Add:Long	192838016	188321840	2.34	(adds/s)
Arith:Add:Float	1484273.1	1482983.4	0.09	(adds/s)
Arith:Add:Double	1485996.2	1481910.2	0.27	(adds/s)
Arith:Mult:Int	116997232	116747128	0.21	(multiplies/s)
Arith:Mult:Long	59741112	60856160	-1.87	(multiplies/s)
Arith:Mult:Float	1522337	1519513.2	0.19	(multiplies/s)
Arith:Mult:Double	1464426.1	1457029	0.51	(multiplies/s)
Assign:Same:Scalar:Local	2362722050	2344758530	0.76	(assignments/s
Assign:Same:Scalar:Instance	911725950	917389310	-0.62	(assignments/s
Assign:Same:Scalar:Class	737395200	736152770	0.17	(assignments/s
Assign:Other:Scalar:Instance	295890272	296040640	-0.05	(assignments/s
Assign:Other:Scalar:Class	268851840	267262064	0.59	(assignments/s
Cast:IntFloat	50712684	50638232	0.15	(casts/s)
Cast:IntDouble	50532808	50759816	-0.45	(casts/s)
Create:Array:Int:16	7181555	6869601.5	4.34	(arrays/s)
Create:Array:Int:32	4960039	4688108	5.48	(arrays/s)
Create:Array:Long:1	9575687	9376753	2.08	(arrays/s)
Create:Array:Long:2	9238229	9014085	2.43	(arrays/s)
Create:Array:Long:64	1286189.8	1214997.6	5.54	(arrays/s)
Create:Array:Long:128	717790.56	672445.5	6.32	(arrays/s)
Create:Array:Object:1	9900894	9530014	3.75	(arrays/s)
Create:Array:Object:2	9549455	9252315	3.11	(arrays/s)
Create:Array:Object:4	9270639	9013093	2.78	(arrays/s)
Create:Array:Object:8	8566349	8227791	3.95	(arrays/s)
Create:Object:Simple	2048000	1788802.5	12.66	(objects/s)
Create:Object:Simple:Constructor	2032149.2	1788802.5	11.97	(objects/s)
Create:Object:Simple:1Field	1933169.8	1709515.9	11.57	(objects/s)
Create:Object:Subclass	2038216.5	1764908.6	13.41	(objects/s)
Create:Object:Complex	1678138.4	1481374.4	11.73	(objects/s)
Exception:New	196439.53	141303.75	28.07	(exceptions/s)
Exception:Method	185749.53	135227.94	27.20	(exceptions/s)
Loop:For	511600320	505679008	1.16	(iterations/s)
Loop:ReverseFor	511201248	482592032	5.60	(iterations/s)
Math:AbsLong	37215220	37261768	-0.13	(operations/s)
Math:AbsDouble	31678268	31813592	-0.43	(operations/s)
Math:MaxFloat	28885754	28957228	-0.25	(operations/s)
Math:MaxDouble	29534024	28959788	1.94	(operations/s)
Math:MinLong	33553144	31863088	5.04	(operations/s)
Math:MinDouble	29999084	28429638	5.23	(operations/s)
Math:SinDouble	8395163	8399036	-0.05	(operations/s)
Math:CosDouble	7619756.5	7624011	-0.06	(operations/s)
Math:AtanDouble	4082934.5	4059867.2	0.56	(operations/s)
Math:Atan2Double	3620613.5	3464433.8	4.31	(operations/s)
Math:FloorDouble	4026740	3768169.2	6.42	(operations/s)
Math:PowDouble	750348.06	788845.25	-5.13	(operations/s)
Math:RintDouble	4026344.2	3810941.5	5.35	(operations/s)
Math:RoundFloat	2788671	2745308.2	1.55	(operations/s)
Math:IEEERemainderDouble	422791.1	422895.84	-0.02	(operations/s)
Method:Same:Instance	159960944	158750080	0.76	(calls/s)
Method:Same:SynchronizedInstance	5423729	5422652	0.02	(calls/s)
Method:Same:FinalInstance	174646240	168646416	3.44	(calls/s)
Method:Same:Class	175735072	175511520	0.13	(calls/s)
Method:Other:Instance	31480450	31538018	-0.18	(calls/s)
Method:Other:InstanceOfAbstract	31528914	31538018	-0.03	(calls/s)
Method:Other:Class	39130644	39163380	-0.08	(calls/s)
Average of *all* JGF microbenchmarks			3.01	

Table 2: Benchmark application results

Benchmark	Original	TCH	Difference (%)	Units
SPECjbb	6727	6524	3.02	(Operations/s)
Series	488.05	533.15	-9.24	(Coefficients/s)
LUFact	192.70	189.64	1.59	(Mflops/s)
HeapSort	634678.90	632111.25	0.40	(Items/s)
Crypt	2235.01	2239.02	-0.18	(Kbyte/s)
FFT	129814.42	138070.45	-6.36	(Samples/s)
SOR	16.84	16.14	4.15	(Iterations/s)
SparseMatmult	12.72	12.76	-0.24	(Iterations/s)
Euler	4.44	4.67	-5.18	(Timesteps/s)
MolDyn	181487.31	187443.61	-3.28	(Interactions/s)
MonteCarlo	406.16	277.48	31.68	(Samples/s)
RayTracer	1183.73	1249.95	-5.59	(Pixels/s)
AlphaBetaSearch	798061.56	798356.90	-0.04	(Positions/s)

gory are short codes that carry out specific operations frequently used in Grande applications. The third category consists of large scale applications that demonstrate Java's potential in tackling real problems.

With few exceptions, the comparison shows that the performance of the rewritten bytecodes is close to their original performance. Tables 1 and 2 summarize the results. The former presents the throughput of microbenchmarks from the first section of the JGF benchmark suite. The latter presents the results of SPECjbb and of the remaining applications from the JGF benchmark suite. The "Difference" column shows the difference between the original and instrumented benchmarks. Let A and B be the throughputs of the original benchmark and its TCH version respectively. The corresponding value in the "Difference" column is $100*(A-B)/A$. Consequently, a positive value indicates higher throughput of the original application.

The most significant performance difference in Table 1 is observed for object creation benchmarks (denoted by the prefix "Create:Object"), and exception benchmarks (denoted by the prefix "Exception"). In both cases, the difference is caused by the increased cost of creating rewritten objects. This increased cost is due to the fact that these objects' inheritance chain is augmented with an additional class at the top of the hierarchy (`java.lang.Object` or `java.lang.Throwable`; see Figure 1). As a result, an additional constructor needs to be called during their creation.

The instrumentation overhead in SPECjbb (Table 2) is only 3%. The most significant throughput difference among the macrobenchmarks is observed in the Monte Carlo benchmark, whose throughput decreases by 32% as a result of the instrumentation. The Monte Carlo benchmark extensively uses the class `java.util.Random`, whose methods are often inlined by the just-in-time compiler (JIT). However, the JIT does not inline the counterparts of these methods in the instrumented code, which causes a decrease in performance.

5. APPLICATIONS OF TCH

We have employed TCH in Java to implement two bytecode instrumentation-based systems: (i) a memory profiler, and (ii) a distributed runtime for Java. In both systems the instrumentation is performed using the Bytecode Engineering Library (BCEL) [9].

5.1 Memory profiler

Our memory profiler is a tool for gathering memory allocation statistics. It can be used to explore memory usage and detect memory leaks in any Java program. The bytecode instrumentation is performed dynamically, by intercepting the class loading process with the BCEL custom class loader.

The profiling transformation intercepts all bytecode instructions used in object and array creation (*i.e.*, `new`, `newarray`, `anewarray`, and `multinewarray`). It also intercepts the system API calls that create new object instances, *i.e.*, `java.lang.Object.clone()`, and `java.lang.reflect.Constructor.newInstance(Object[])`. In the bytecode, after each such *allocation event*, the profiler transformer inserts a call to a special handler. This handler may record any interesting data associated with the event, *e.g.*, the class of the created object, its size, the time of its creation, *etc*. (The newly created object is passed as a parameter to the handler.) The profiling transformation augments each class with a method that returns its instance size.

The profiler handler accesses the internal profiler logic, which is implemented in pure Java. The implementation of the profiler logic extensively uses system classes, *e.g.*, `java.lang.System`, `java.util.Hashtable`, `java.util.LinkedList`, `java.util.Iterator`, `java.io.PrintStream`, and `java.io.FileOutputStream`. For example, an instance of class `java.util.Hashtable` is used for mapping between a class name and a counter of allocated class instances. An instance of `java.util.LinkedList` is used to record the times of allocation events. Instances of class `java.io.FileOutputStream` are used to spool the collected data to files. (The files are used during the execution because the accumulated data may be too large, especially in long-running applications.)

TCH benefits the profiler in two ways. First, it allows the profiler to explore applications that use the same system classes that it uses in its implementation. While the application uses the instrumented system classes, the profiler logic employs their original counterparts. Thus, the original functionality of system classes remains available to the profiler. Without TCH, the profiler would have to use the instrumented system classes, which would result in infinite recursion. Second, TCH enables the profiler to instrument

Table 3: Profiler output – object and array allocations in SPECjbb

Class	User classes		System classes	
	count	%	count	%
char[]	567268	22.27	1979950	77.73
TCH.java.lang.String	8099041	85.35	1390047	14.65
TCH.java.util.Hashtable$Entry[]	0	0.00	76188	100.00
TCH.java.util.Hashtable$Entry	0	0.00	38097	100.00
int[]	76235	99.65	270	0.35
java.lang.Object[]	700428	99.98	163	0.02

all system classes (dynamically). Consequently, it can collect more accurate results. If the profiler instrumented only user classes, then allocations performed within the system classes would not be detected. If system classes are not modified, then allocations of arrays and system class object instances that are performed within the code of system classes are impossible to intercept. By contrast, allocations of user class instances can still be intercepted by modifying their constructors.

Table 3 illustrates the importance of the second feature in profiling of the SPECjbb benchmark. The table presents the final values of creation counters of several classes used by the benchmark. The columns "User classes" and "System classes" indicate the number of instances created in user classes and system classes respectively. The table shows that a large number of allocations occur within the code of the system classes. For example, most character arrays (`char[]`) are allocated within the system classes. Moreover, the creation of the system classes `java.util.Hashtable$Entry[]` and `java.util.Hashtable$Entry` occurs only within the system code. Without the ability to instrument system classes, all these allocations would remain undetected.

5.1.1 Related systems

The Cougaar Memory Profiler (CMP) [3] is a bytecode instrumentation-based tool for memory profiling of Java programs. The developer selects which classes should be tracked and runs an automated class file editor (using BCEL) to add profiling instructions to the constructors. The profiler maintains pointers to the live instances and can display various useful debugging information, *e.g.*, the total number of allocations of a profiled class, including the number of live and garbage-collected instances. In contrast to our memory profiler, the instrumentation is performed statically, *i.e.*, before the execution.

Until recently, CMP did not have any support for profiling Java system classes. Currently, it allows the user to (statically) transform the classes in system packages. The instrumented classes are loaded into the JVM by means of the command line option *-Xbootclasspath*, which allows an alternative location of the system classes to be specified when starting the JVM.

The CMP manual advises users to avoid modifying system classes as much as possible in order to prevent the potential loading errors that occur if the profiler's code uses system classes that are being analyzed. The manual states that *"... a call to '`new HashSet()`' will fail if `HashSet` is profiled, due to a stack overflow caused by the circular reference. Similarly, if all of `java.lang` will be profiled, then `Strings` should be carefully handled to avoid string allocations, including any calls to `System.out`."* By contrast, TCH allows our memory profiler to easily analyze any system class, even when this system class is used in the profiler's logic.

5.2 Instrumenting Java bytecode for distributed execution

We have employed the TCH approach to implement a portable distributed runtime for multithreaded Java, which we call JavaSplit [11]. JavaSplit uses bytecode instrumentation to transparently distribute the threads and objects of a standard Java application among the available machines. The instrumentation intercepts events that are interesting in the context of distributed execution, such as thread creation, accesses to shared data, and synchronization. Shared data is managed by an object-based distributed shared memory (DSM). All the runtime logic, including DSM, is implemented in pure Java. Therefore, each node carries out its part of the distributed computation using nothing but its local standard (unmodified) JVM. JavaSplit employs IP-based communication, accessing the network through the standard Java socket interface.

The distinguishing feature of JavaSplit is its portability. The use of standard JVMs in conjunction with IP-based communication allows virtually any commodity workstation to join JavaSplit. Moreover, a new node does not need to install any software or hardware. It needs only to receive the application bytecode and the runtime modules (both of which it can get by means of the customizable class loading mechanism).

The correctness and consistency of the JavaSplit system depends on its ability to intercept various events such as accesses to shared data, lock operations, *etc.* Therefore, JavaSplit must be able to instrument any Java class required by the original application, including the Java system classes. Using TCH enables JavaSplit to achieve this goal by alleviating the difficulties of transforming Java system classes, as described in Section 3.

Many system classes are used in the implementation of the JavaSplit runtime modules. In particular, we make extensive use of classes from the `java.util`, `java.io`, and `java.net` packages. The `java.util` package provides JavaSplit modules with complex data structures. The other two packages are used for I/O and networking. TCH enables the original versions of these classes, which are used by the runtime modules, to coexist with their instrumented counterparts, which are required by the application.

Without TCH we would have had to use the instrumented versions of system classes in the runtime modules, or else

avoid using system classes in the runtime modules completely. Neither option is practical. The former would result in erroneous behavior of the runtime modules, *e.g.*, infinite recursion. The latter would require reimplementing the data structures from `java.util`, as well as the I/O and networking facilities. While reimplementing data structures from `java.util` would be a long, tedious task, reimplementing the I/O and networking would require incorporating user-defined native methods in the runtime code, thus compromising portability.

5.2.1 Related systems

Like JavaSplit, J-Orchestra [16] partitions Java applications for distributed execution through bytecode instrumentation. However, the goal of partitioning is different. While JavaSplit creates a supercomputer from interconnected commodity workstations, J-Orchestra aims to split a centralized application into distinct entities running on the most functionally suitable sites. For example, J-Orchestra may execute a computation intensive application with a GUI on two machines: one with a fast processor and another with a graphical screen. Unlike JavaSplit, which employs an object-based distributed shared memory and monitors accesses to the shared data, J-Orchestra uses proxies to access remote objects. It substitutes method calls and direct object references with remote method calls and proxy references respectively.

The key difference between the two systems is in the treatment of Java system classes with native dependencies (*i.e.*, classes that have native methods or can be accessed from such classes). In J-Orchestra they are perceived as *unmodifiable code* and therefore cannot be rewritten to access remote objects through a proxy. This results in certain constraints on the data placement. All instances of a system class with native dependencies are placed on the same node. Moreover, any class that can be referenced from it must also be placed on that node. Due to the strong class dependencies within Java packages, this usually results in partitions that coincide with package boundaries. In contrast, JavaSplit supports arbitrary partitioning because TCH allows even system classes with native dependencies to be rewritten for distributed execution.

Addistant [15] is yet another system that partitions Java programs. Like J-Orchestra, it aims at functional distribution, rather than high performance computing. It instruments Java bytecode at load-time using the Javassist [7] framework. Like J-Orchestra, it employs the remote proxy model to bridge between objects on different nodes. As a result of the difficulties described in Section 3.2, Addistant is unable to transform system classes at load-time. Therefore, it introduces several bytecode rewriting workarounds, the applicability of which depends on the type of interaction between the classes. The creators of Addistant admit that even if all system classes could, like user classes, be rewritten, it would still be hard to modify them consistently, since *"...certain runtime systems such as a system class loader depend on the definition of the system classes."* This is additional evidence of the circular dependency introduced by the requirement to use a single (instrumented) version of a class. The use of TCH would allow Addistant to modify all system classes dynamically while avoiding the infinite recursion problem.

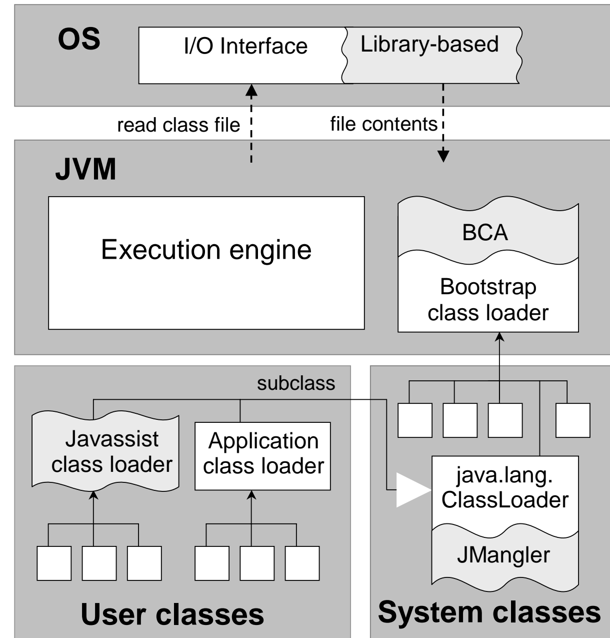


Figure 8: Summary of existing interception techniques for JVM

6. RELATED INSTRUMENTATION APPROACHES

The key idea in the TCH approach is creation of a hierarchy of instrumented classes isomorphic to the original one. This idea has been employed (for subsets of classes) in the context of application partitioning [15, 16], mainly to implement hierarchies of proxies used to access remote objects. The inheritance relationship between the proxy classes is made identical to that of the original classes in order to allow a variable of type `proxy_of_A` to contain a reference to a proxy of any subclass of `A`.

J-Orchestra [16], already mentioned in Section 5.2.1, has an instrumentation approach closest to that of TCH. It creates class hierarchies that mimic the structure of the original one. It differs from TCH mainly in that each isomorphic hierarchy is created only for a subset of classes. For two system classes with native dependencies, `A` and its subclass `B`, J-Orchestra generates two proxies, `anchored.A` and `anchored.B`, the latter subclassing the former. For other classes, J-Orchestra makes the names of the proxy classes identical to the original class names, whereas the names of the real application classes are augmented with the suffix "`__remote`". For example, two classes, `A` and its subclass `B`, are renamed `A__remote` and `B__remote`, the latter subclassing the former.

7. LOAD-TIME INSTRUMENTATION FRAMEWORKS

In this section we overview the existing frameworks for dynamic instrumentation of Java bytecode, focusing on their ability to intercept system classes. Figure 8 illustrates the various approaches to class interception. Our main observation is that, without the use of TCH, only nonportable techniques allow instrumenting system classes at run time.

7.1 Custom class loaders

The Java class loading mechanism allows users to install custom class loaders to intercept class files at load-time. A custom class loader must subclass the system class `java.lang.ClassLoader`. This strategy is employed by Javassist [7], JOIE [8], and BCEL [9]. The applicability of this approach is limited to applications that do not use their own custom class loaders. This is because only one class loader can affect the definition of a class when it is being loaded. Without TCH, a custom class loader cannot intercept all system classes because of the difficulties described in section 3.2, such as the preloaded classes problem.

7.2 JVM dependent interception

The behavior of the class loading mechanisms can be affected by modifying the implementation of the JVM. Binary Component Adaptation (BCA) [12] introduces an adaptation module for transforming the internal JVM data structure that represents a loaded class. Unlike most instrumentation frameworks, BCA allows the system classes to be redefined. Unfortunately BCA requires a custom JVM, thus compromising portability.

7.3 Library-based interception

Duncan and Hölzle [10] introduce *library-based load time adaptation.* They intercept and modify the class files as they are being fetched from the file system. This is achieved by modifying a dynamically-linked standard library that is responsible for reading files. Like BCA, this approach allows instrumentation of system classes at the expense of portability. It requires that a custom DLL be provided for every operating system.

7.4 Class loader independent interception

JMangler [13] provides a portable interception facility, which, unlike Javassist, BCEL and JOIE, allows the application to use custom class loaders. This is achieved by providing a modified version of the final method `defineClass()` in the class `java.lang.ClassLoader`. Because the modified behavior is enforced for every subclass of `java.lang.ClassLoader`, JMangler is activated whenever an application-specific class is loaded. In contrast to BCA and DLL-based load-time adaptation, this approach is limited because it cannot transform system classes without employing TCH.

8. CONCLUSION

We have presented TCH, a general instrumentation strategy capable of instrumenting system classes while avoiding the associated pitfalls. In contrast to those few approaches [10, 12] that allow arbitrary transformations of system classes, TCH does not modify the language infrastructure and is therefore portable. Most existing instrumentation-based systems do not transform system classes, thus making an unnatural distinction between user-defined and system classes. Consequently, these systems either invest considerable effort in finding context-specific solutions to overcome their inability to transform system classes, or limit their functionality to user classes. The TCH approach provides these systems with an opportunity to overcome their limitations and find simpler and more efficient ways to achieve their goals without compromising portability.

9. REFERENCES

[1] The AspectJ home page. http://eclipse.org/aspectj/.

[2] The AspectWerkz home page. http://aspectwerkz.codehaus.org.

[3] The Cougaar Memory Profiler home page. http://cougaar.org/projects/profiler/ or http://profiler.cougaar.org.

[4] The JBoss AOP project home page. http://www.jboss.org/developers/projects/jboss/aop.jsp.

[5] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000/.

[6] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[7] S. Chiba. Load-time structural reflection in Java. *Lecture Notes in Computer Science*, 1850:313–336, 2000.

[8] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[9] M. Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[10] A. Duncan and U. Hölzle. Load-time adaptation: Efficient and non-intrusive language extension for virtual machines. *Tech. Rep. 99-09, Department of Computer Science*, February 1999.

[11] M. Factor, A. Schuster, and K. Shagin. JavaSplit: A runtime for execution of monolithic Java programs on heterogeneous collections of commodity workstations. In *IEEE Fifth Int'l. Conference on Cluster Computing*, pages 110–119, Hong Kong, December 2003.

[12] R. Keller and U. Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445, 1998.

[13] G. Kniesel, P. Costanza, and M. Austermann. JMangler – a framework for load-time transformation of Java class files. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2001.

[14] T. Lindholm and F. Yellin. *The JVM Specification, 2nd edition.* Addison-Wesley, 1999.

[15] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. *Lecture Notes in Computer Science*, 2072, 2001.

[16] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, June 2002.