JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 11
Special issue: OOPS track at SAC 2004, Nicosia/Cyprus

SmartReflection: Efficient Introspection in Java

Walter Cazzola
DICo - Department of Informatics and Communication,
Università degli Studi di Milano
cazzola@dico.unimi.it

In the last few years the interest in reflection has grown and many modern programming languages/environments (e.g., Java and .NET) have provided the programmer with reflective mechanisms, i.e., with the ability of dynamically looking into (introspect) the structure of the code from the code itself. In spite of its evident usefulness, reflection has many detractors, who claim that it is too inefficient to be used with real profit. In this work, we have investigated about the performance issue in the context of the Java reflection library and presented a different approach to the introspection in Java that improves its performances. The basic idea of the proposed approach consists of moving most of the overhead due to the dynamic introspection from run-time to compile-time. The efficiency improvement has been proved by providing a new reflection library compliant – that is, it provides exactly the same services –, with the standard Java reflection library based on the proposed approach. This paper is focused on speeding up the reification and the invocation of methods, i.e., on the class `SmartMethod` that replaces the class `Method` of the standard reflection library.
Keywords: Reflection, Introspection, Optimization, Java, Java Reflection Library.

1 INTRODUCTION

In the last few years many researchers (see for example [15,3]) stressed the relevance of reflection, reflective behavior and meta-level architectures. This growing interest in reflection is also testified by the fact that both Java [1] and .NET [8] architectures, – that is, two of the most used programming architectures – are intrinsically reflective [12] or provide the programmer with many reflective features (see the Java core reflection library [13] and [6] for an overview of the reflective features of Java).

Reflection is defined as the activity performed by a program when doing computations about itself [10]. The reflective activity is incarnated by several different activities: the most relevant and used are *introspection* and *intercession*. They are defined as the ability for a program to respectively observe and modify its own structure, state and execution. The reflective activities are usually carried out on representatives (called *reifications*) of the program itself constantly kept updated with their referent, i.e., any variation in the program structure and behavior is reified into its reifications and vice versa.

Modern programming languages/environments such as Java and .NET provide the programmer with limited reflective capabilities as a mix of introspection and

Cite this article as follows: Walter Cazzola: "SmartReflection: Efficient Introspection in Java", in *Journal of Object Technology*, vol. 3, no. 11, December 2004, Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, pp. 117–132, http://www.jot.fm/issues/issue_2004_12/article6.

partially of intercession (intercession is usually limited to method invocation and attribute manipulation). Generally, the programmer can dynamically reify some structural aspects of his program as methods, constructors, attributes, and classes (i.e., (s)he can observe the program structure); then (s)he can discretionary use such reifications for invoking methods and creating new objects (i.e., (s)he can modify the program behavior). Unfortunately, coming into the limelight both merits and flaws of reflection are more evident. The most raised issue against the use of a reflective solution is related to its performance. Obviously, introspection and intercession are expensive tasks when carried out during program execution. Many attempts have been done to improve this situation, most of them are related to move reflection from run-time to compile-time [4,11] or load-time [5]. But, there are still many situations where the introspection and intercession must take place at run-time. e.g., in remote communications where we are looking for unknown services.

In this work, we have investigated how to speed up the introspection and intercession capability provided by the Java reflection library. In particular, we have proposed a different approach to reification that moves most of the overheads due to the reflective activity from run-time to compile-time. The approach consists of demanding the reflective activities to a reification stub prepared at compile-time. Such an approach also allows of speeding up the reflective method invocations whose performance usually presents penalties due to the dynamic resolution of many details that could be statically solved, e.g., type checking. The whole paper focuses on applying such an approach to the methods reification and on their invocation, i.e., we will explore the implementation of the class `SmartMethod` that provides the same functionality of the corresponding class `Method` of the Java reflection library but improving the efficiency of method invocation. The work is based on the Java language, and on the efficiency of its reflection library [13], but similar considerations apply to the .NET reflective mechanism as well.

The rest of the paper is organized as follows. Section 2 shows the basic idea for optimizing the Java method `invoke()`, whereas section 3 goes deepen in the realization. Section 4 gives a glance at the performance improvements. Finally in section 5 we draw our conclusions and propose some future works.

2 UNFOLDING METHOD LOOKUP

The Java core reflection library [13] provides the programmer with classes (`Field`, `Method`, and so on) whose instances reify specific aspects (that is, respectively, fields, methods and so on) of a class. Such aspects are reified by invoking specific methods (e.g., `getFields()` for reifying all the fields of a class) on a reification of a Java class (that is, an instance of the class `Class`).

In this work, we focus our attention on the class `Method`. Each of its instances reifies all data related to a given method (i.e., its name, its argument types, its return type and so on) and it also allows the invocation of the reified method (through the

method `invoke()`). As an example, the reflective invocation of the method:

```
public void testMethod(float f);
```

defined by the class `TestClass` is carried out by the following snippet of code:

```
Class c = Class.forName("TestClass");
Method m = c.getMethod("testMethod", new Class[]{Float.TYPE});
m.invoke(c.newInstance(), new Object[]{new Float(7)});
```

This piece of code is quite familiar to Java programmers and it is deeply rooted in the reflection terminology. Initially, the class `TestClass` must be reified, then such a reification can be inspected (*introspection*) and the method `testMethod()` can be reified, e.g., by invoking the method `getMethod()`. Finally, `m` is an instance of the class `Method` that represents a reification of the requested method, `m` has the reflective ability of activating the method it represents by invoking the method `invoke()`.

As explained in the API documentation, the method `invoke()` activates the underlying method reified by **`this`** (an instance of `Method`), on the specified object with the specified parameters. Individual parameters are automatically unwrapped to match primitive formal parameters, and both primitive and reference parameters are subjected to method invocation conversions as necessary. The underlying method is invoked using dynamic method lookup (cf. [7] section 15.12.4.4); in particular, dynamic dispatching based on the run-time type of the target object will occur.

The described approach implies that a lot of execution time is spent for simulating the method lookup, for checking the method compatibility and, above all, for dispatching the method call to the referring object in accordance with the fact that the method is inherited, invoked through an interface or invoked by exploiting the late binding mechanism. This fact is fairly evident comparing the time spent in invoking a method through a class with the time spent by using an interface (see table 1).

Our idea for speeding up method invocation is quite simple and consists of letting the compiler resolve method overloading for us and delegating the real invocation to the standard invocation mechanism (not to the reflective one).

To realize this mechanism, we have stolen the *stub* idea from the Java RMI [14]. Each class, that we refer as *referent class*, is associated with another class, named after the referent class name by adding the suffix `"_InvokeStub"`, that plays the role of a stub. The stub implements an *ad hoc* method `invoke()`, called `smartInvoke()`,

tailored on the referent class and implementing also some ancillary routines. The method `smartInvoke()` has a per-class structure that renders the method dispatch more efficient providing a binding for each method defined by the referent class to its direct invocation.

At this point, the reflective method `invoke()`, when used to call a method of the referent class, simply delegates the execution to the method `smartInvoke()` of the associated stub and indirectly to the direct call of the method that we would like to invoke. In this way, the reflective invocation exploits the standard invocation mechanism taking benefit from static method lookup and dispatching.

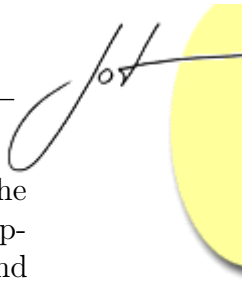
Reconsidering our example, the class `TestClass` will be associated with the class `TestClass_InvokeStub`. Such a class will implement the method `smartInvoke()` providing a mapping to a direct call for each method defined or inherited by the class `TestClass` itself; in particular it provides the following mapping for the method `testMethod()`:

```
public class TestClass_InvokeStub extends SmartInvokeStub {
  public Object smartInvoke(Object o, int h, Object[] a){
    switch (h) {
      case -1822788606:
        ((TestClass)o).testMethod(((Float)a[0]).floatValue());
        return null;
        /* similarly, the omitted code would have shown the case state-
           ments for invoking the other methods defined or inherited by
           the class TestClass. */
    } /* end switch */
  }
}
```

The method `smartInvoke()` basically consists of a **`switch`** statement indexed on the hashcode of the methods (static or not) that in somehow can be called through the class (static methods), through an interface or through an instance of the class. This means that it is indexed on all **`public`**, **`protected`**, default-access and **`private`** methods both defined and inherited by the class. The fact that private methods are included could sound weird but the Java core reflection library permits, thanks to the method `setAccessible()`, a meta-programmer to circumvent the visibility modifier declared by the programmer.

3 METHOD OPTIMIZATION

The described idea is embedded in a more pretentious project, called *SmartReflection*, consisting of optimizing the whole Java core reflection library. The optimization of method invocation has been realized by: (i) writing a class, named `SmartMethod`,



which exploits the described mechanism and (ii) a generator which unfolds the method invocation and builds the described stubs. Similarly, all the classes supporting reflection in the standard Java reflection library (i.e., `Class`, `Field`, and `Constructor`) have been replaced by compliant but more efficient classes (respectively, `SmartClass`, `SmartField`, and `SmartConstructor`) based on the same idea exposed in this work. In the rest of the section we describe the class `SmartMethod` and its interface, how the generator works and how the methods are reified in instances of the class `SmartMethod`.

`Method` vs `SmartMethod`

We have built a class, named `SmartMethod` which provides the programmer with exactly all the functionality provided by the `Method` class. This means, as shown in the code below, that the public interface of the class `SmartMethod` defines the same methods, with the same signature as the class `Method`.

```
package cazzolaw.lang.reflect;

import java.lang.reflect.Member;
import java.lang.reflect.AccessibleObject;

public class SmartMethod extends AccessibleObject implements Member{
   /* SmartMethod constructor */
  public SmartMethod(String _methodName, Class[] _argClasses,
            Class _declaringClass, SmartInvokeStub _stub) {...}
   /* Methods inherited from class AccessibleObject */
  public static void setAccessible(AccessibleObject[] a,
      boolean flag) throws SecurityException {...}
  public void setAccessible(boolean flag)
      throws SecurityException {...}
  public boolean isAccessible() {...}
   /* Methods required by the interface Member */
  public Class getDeclaringClass(){...}
  public int getModifiers() {...}
  public String getName() {...}
    /* Methods defined as in the class Method */
  public boolean equals(Object obj) {...}
  public Class[] getExceptionTypes() {...}
  public Class[] getParameterTypes() {...}
  public Class getReturnType() {...}
  public int hashCode() {...}
  public toString() {...}
```

```
  public Object invoke(Object obj, Object[] args)
    throws InvocationTargetException,
      IllegalArgumentException,IllegalAccessException {...}
}
```

The class SmartMethod, as well as the class Method, must provide a general approach to method reification, therefore it cannot be statically bound to the stubs and it cannot directly embed them. Thus, the class SmartMethod is associated to the right stub by a dynamic clientship that is perfected when the method is reified, that is, when an instance of the SmartMethod class is created.

On the contrary of the class Method, the class SmartMethod has a public constructor whose unique aim consists of associating the method reification with the corresponding stub. The constructor knows which is the declaring class of the method to be reified (information passed to the constructor of the class Method as well) from this information it is able to determine which is the stub tailored on such a class and to get an instance of such a stub.

```
public SmartMethod(String _methodName, Class[] _argClasses,
          Class _declaringClass, SmartInvokeStub _stub) {
 methodName = _methodName;
 parameterTypes = _argClasses;
 declaringClass = _declaringClass;
 this.stub = _stub;
 code = stub.hashCode();
 returnClass = stub.smartGetReturnType(code);
 _modifiers = stub.smartGetModifiers(code);
 _exceptions = stub.smartGetExceptionsType(code);
 _accessible = true;
}
```

Notwithstanding that the class has a public constructor, instances of the class should be got by inspecting the class reification itself, in fact, as visible in the code of the constructor above, the stub is passed as an argument and it could be difficult for the programmer to create the correct one without the API support.

The stub provides the method reification with a connection to the low-level method invocation unfolding. This separation grants the flexibility of the approach because free the implementation of the SmartMethod class from the knowledge of the static type of the caller but it is also one of its flaws, because we invoke stub's methods by exploiting late binding and therefore by resolving method dispatch to the stub at run-time.

```
public Object invoke(Object obj, Object[] args) {
  return stub.smartInvoke(obj, code, args);
}
```

The method `invoke()`, whose code is reported above, is just a wrapper for an invocation to the method `smartInvoke()` defined in the stub and connected at construction-time. Similarly, all information about the method (return type, exception types, and so on) are encapsulated in the stub (thanks to the ancillary routines abovementioned) and can be retrieved, when the method is reified, efficiently as well.

The class `SmartMethod` has been designed to provide the programmer with the same functionality and the same constraints of the class `Method`. Therefore, this means that we can reify the same category of methods both using `Method` and `SmartMethod` and the reflective invocation carried out by an instance of `SmartMethod` can be inhibited by revoking the `ReflectPermission` through a security manager as well.

Stub Generation

The automatic generation of stubs represents the most important and also the most delicate and time-consuming activity to carry out to grant the correct behavior of our mechanism. In our idea, static generation, i.e., at compile-time, is the best choice in term of performance to build the stubs but this solution has also some drawbacks: i) at compile-time it is difficult to know how many classes need a stub and probably will be generated too much stubs, but above all ii) there are situations in which neither the bytecode nor the source code of the classes to reify are available at compile-time, e.g., in the case of classes generated on-the-fly or dynamically downloaded from remote (RMI stubs and applet classes).

We have chosen a compromise between static and dynamic stub generation: i) providing a tool, named `SmartInvokeC`, for the automatic generation of the stub from the class bytecode at compile-time, and ii) enabling the class `SmartClass`, who is responsible of the method reification, of detecting the absence of the stub bytecode and of generating it on-the-fly. This compromise permits of obviating to the issues raised above rendering the approach applicable in every context.

The `SmartInvokeC` inspects the bytecode of a class looking for information about, among the others, the methods that are invocable, that is, as said before, all the methods, independently of their right accesses, declared by one of the classes in the class hierarchy of the inspected class. Inspection takes place recursively on each class c in the class hierarchy by collecting all its declared methods; this raking of methods is carried out by exploiting the standard reflective mechanism

(i.e., the `getDeclaredMethods()` method) retrieving all the methods (both **public**, **protected**, default-access and **private**) declared in the class (see the code of the `getMethodsList()` method, reported below).

The Java method invocation mechanism establishes that a method matching the signature of the invoked method whose visibility is not hidden by the definition of another method with the same signature is activated. Therefore, it is activated the method implemented by the dynamic type of the caller or by one of its ancestors (the first occurrence encountered following the inheritance link) when a method is invoked. This behavior is also granted with our reflective invocation by removing the overridden methods from the list of the invocable methods and delegating their activation to the dynamic lookup provided by the direct invocation of Java.

Methods collection and classification is realized by the following snippet of code:

```
protected Method[] getMethodsList() {
  Class _class = Class.forName(ClassName);
    /* ClassName is the name of the class we are introspecting. */
  Package _package = _class.getPackage();
  Package _spackage = _package;
  Method[] _ms;
  Vector _vm = new Vector(), _vmp = new Vector();
  while (_class != null) {
    _ms = _class.getDeclaredMethods();
    for(int i=0;i<_ms.length; i++)
      if !(_vm.contains(_ms[i]) || _vmp.contains(_ms[i]))
       if Modifier.isPrivate(_ms[i].getModifiers()) ||
        (Modifier.isProtected(_ms[i].getModifiers()) &&
          (!isTheSamePackage(_package, _spackage)))
            _vmp.add(_ms[i]);
       else _vm.add(_ms[i]);
    _class = _class.getSuperclass();
    if (_class != null) _spackage=_class.getPackage();
  }
  Method[] _ml = new Method[_vm.size()+_vmp.size()];
  _publicMembers = _vm.size();
  for(int j=0; j<_vm.size(); j++)
    _ml[j] = (Method)_vm.elementAt(j);
  for(int j=0; j<_vmp.size();j++)
    _ml[j+_publicMembers] = (Method)_vmp.elementAt(j);
  return _ml;
}
```

Collected methods are classified in two groups after their access rights. The for-

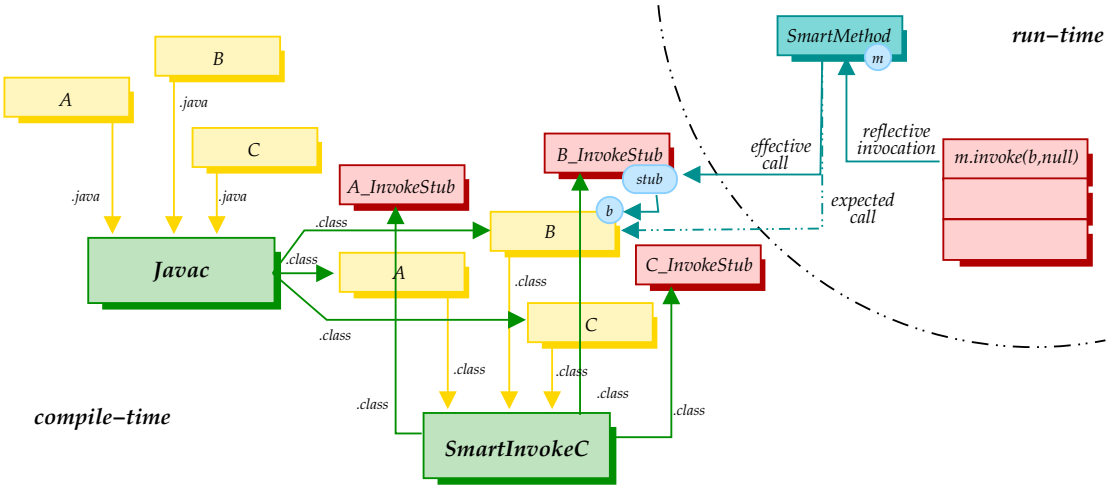


Figure 1: **Compiling the stubs and reflective method invocation through the stubs**

mer group collects the **public** and default-access methods and the methods defined as **protected** in the same package of the examined class; whereas the second collects the remaining methods. We distinguish the methods in these two categories because the methods in the first category can be invoked without restrictions. On the contrary, the invocation of the methods in the second category must obey to some restrictions due to their access qualification. We have gone round such restrictions by delegating their invocation to the Java Native Interface (JNI) [9]. C/C++ programs can invoke Java methods without undergoing to the access restrictions defined by the Java class. Notwithstanding that to find an application for the reflective invocation of **private** methods is not trivial, we have implemented it for compatibility with the standard reflective invocation mechanism.

The whole mechanism for fast retrieving and invoking a method is based on hashcoding the information necessary to discriminate such a method from the others. The method signature (that is, its name and the classes of its arguments) contains all the necessary information for discriminating methods. We do not consider the method return type because it does not contribute to the overloading resolution (that is Java does not allow the definition of two methods with the same name and arguments but different return type). The hashcodes associated with each method are calculated by the `SmartInvokeC` tool during the bytecode analysis. All the hashcodes are calculated from the method signature and embedded in the stub (in particular, in the method `hashCode()` of the stub). Conflicts are managed by associating a manually incremented sub-hashcode with the calculated one. At method reification the corresponding hashcode is retrieved from the stub. This mechanism guarantees the unicity of the hashcode, its fast retrieving and therefore a perfect and fast mechanism for discriminating the invocable methods in the stub.

Figure 1 summarizes how compilation takes place and how the reflective method

invocation exploits the stub. As usual, bytecodes are generated from the classes by the Java compiler (e.g., `javac`). From the bytecode of the classes, the `SmartInvokeC` creates and compiles the corresponding stub classes. At run-time, method reification associates the reified method with the stub of the class declaring the reified method. The reflective invocation of a method through the `invoke()` method of the `SmartMethod` class is hijacked to the stub and after that to the class declaring such a method rather than directly to declaring class. The intermediate step permits to transform the reflective call in a direct call as explained in the previous sections.

Method Reification: **SmartClass**

In the standard Java core reflection library, method reification is the result of class inspection – that is, the class `Class` provides some methods (`getMethod()`, `getMethods()`, `getDeclaredMethod()`, and `getDeclaredMethods()`) which look at a class reification for declared and inherited methods –, hence no explicit `Method` creation is neither necessary nor allowed.

Since we want to supply a complete substitution of the `java.lang.reflect` library with exactly the same functionality (both in terms of API interface and use and in terms of methods behavior) we have also provided a replacement for the class `Class` (named `SmartClass`) that takes care, among the other duties, of method reification.

The class `SmartClass` provides exactly the same interface and functionality of the class `Class`. From the point of view of the method introspection, i.e., the main topic of this work, the class `SmartClass` provides two couple of methods: the first couple (`getDeclaredMethod()`, and `getDeclaredMethods()`) takes care of reifying the methods declared by the class independently of their visibility constraints; whereas the second couple (`getMethod()`, and `getMethods()`) takes care of reifying the public methods declared or inherited by the class. Notwithstanding that the behavior of these methods is quite different, their implementation is very similar: they check if the called method can reify the requested method, in the positive case they create the reification (that is, the instance of `SmartMethod` for the requested method) otherwise an exception is raised.

```
package cazzolaw.lang;

import cazzolaw.lang.reflect.*;
import cazzolaw.generator.*;

public class SmartClass {
  private Hashtable _pool;
  private SmartClass(String cn) throws ClassNotFoundException {...};
  static public SmartClass forName(String classname) ... {...}
```

```
  public SmartMethod getDeclaredMethod(String name, Class[] args)
      throws NoSuchMethodException, SecurityException {
    Class stubClass = null; SmartInvokeStub stub = null;
    try {
      stub = _pool.get(_class.getName()+"_InvokeStub");[1]
      if (stub == null)
        stubClass = Class.forName(_class.getName()+"_InvokeStub");
    } catch(ClassNotFoundException cnfe) {
      SmartInvokeGenerator SIG =
          new SmartInvokeGenerator(_class.getName());
      SIG.generating(); stubClass = SIG.jit();
    } catch(Exception e) {e.printStackTrace();}
    try {
      stub = (SmartInvokeStub)stubClass.newInstance();
    } catch(Exception e) {e.printStackTrace();}
    _pool.put(_class.getName()+"_InvokeStub", stub);
    return new SmartMethod(name, args, _class, stub);
  }
  public SmartMethod[] getDeclaredMethods() ... {...}
  public SmartMethod getMethod(String name, Class[] args) ... {...}
  public SmartMethod getMethods() ... {...}
                ...
}
```

The method applicability does not revest any particular relevance whereas the method reification is more interesting. The method reification consists of creating an instance of the class `SmartMethod` and in associating the reification with the correct stub. It is important to note that the association of the stub with the reification implies the loading of its bytecode from the disk, this one is a very time consuming operation, fortunately another consideration permits of cushioning such an overhead: methods reified from the same class share the same stub. To take advantage of this last consideration, the class `SmartClass` keeps a pool of stubs that provides the required stub when already created/loaded. The above reported piece of code shows the structure of the class `SmartClass` and the implementation of the method `getDeclaredMethod()` neglecting to report the details related to the checking of the method applicability.

Summarizing, the quest for the correct stub takes three steps: i) to look for the stub in the pool (this could save an access to the disk), if the necessary stub has not be used before ii) to try loading the bytecode from the disk, if also this fails

[1]Please note that, the name of the class of the stub is built from the name of its referent class by appending the string `"_InvokeStub"`. At the moment, the management of the possible name clashing associated with this solution is out of our scope.

iii) to generate the code for the stub and the related bytecode on-the-fly (methods `generating()` and `jit()`). As described in the previous section, we have decided to generate the stub code and related bytecode on-the-fly when the corresponding class file is not available. The role of *deus ex-machina* played by the class `SmartClass` during the method reification enables it also to detecting the absence of the class file and therefore it is the perfect actor for generating the stub on-the-fly. As shown in the code above, the code generation is carried out by the same routine used by the tool `SmartInvokeC` and the bytecode creation is entrusted to the `instantj` library[2].

4 PERFORMANCE EVALUATION

We have quantified the overall performance of the method `invoke()` of the class `SmartMethod` with respect to standard Java method invocation and the method `invoke()` of the Java core reflection library. The aim of our experiments consists of measuring how long standard Java method invocation and both kind of reflective invocations take to invoke a method. All the experiments were performed on an Intel® P4@2.2 GHz with 512Mb RAM running Linux (kernel version 2.4.21), and `jdk v1.4.2` (last stable and official release at the time of writing).

The scenario of our experiments is composed of a class which implements a simple interface. This class defines some dummy methods; these methods are distinguishable for their access rights (**`public`**, **`protected`**, default-access or **`private`**). Methods taken in consideration for the experiments – following the hints given by the Sun's FAQ on Java HotSpot VM benchmarking[3] –, have an argument, compute some values by using such an argument and return the computed value. In this way, we avoid the optimizations carried out by HotSpot as short method inlining and the removal of dead code that will not render germane the comparison with the direct method call. Basically, the benchmarking has been carried out by repeatedly invoking on a class (or on an interface) these methods and then by calculating the average of the achieved time.

Table 1 summarizes the results of our experiments. The second column represents how long the standard method invocation takes to invoke a **`public`** method. Similarly, the third and the forth columns show how long takes the same invocation carried out by using, respectively, the standard method `invoke()` and our method `invoke()`. The experiments have been done by invoking the methods through a class (second and third rows) and through an interface (fifth and sixth rows) and both enabling and disabling the HotSpot just-in-time compiler.

Calling a method on a class results nearly 25% faster by using our approach than by using the standard method `invoke()`. Notwithstanding this improvement, as expected (see section 2), we got the best by invoking methods on an interface. In this case, our approach is nearly 60% faster than standard method `invoke()`.

[2] See. `http://instantj.sourceforge.net`.
[3] See. `http://java.sun.com/docs/hotspot/PerformanceFAQ.html`.

call via a class	*direct call*	*invoke*	*smart-invoke*
HotSpot	0.0000867	0.000204	0.000154
HotSpot (disabled)	0.0006602	0.0017125	0.0012946
call via an interface	*direct call*	*invoke*	*smart-invoke*
HotSpot	0.0001213	0.0027938	0.0002116
HotSpot (disabled)	0.000606	0.0039545	0.0012208
			* *all the reported time are expressed in milliseconds.*

Table 1: **Direct call, `invoke()` and our smart `invoke()` in comparison.**

However, we are still far from getting the same performance as by using direct invocation.

Table 1, for sake of clarity, summarizes the results only related to the invocation of **`public`** methods. We have achieved quite similar figures invoking **`protected`** and default-access methods. On the contrary, less good results have been achieved by invoking **`private`** methods. This fact can be ascribed to the use of JNI for working around the access protection, a further improvement should be achieved by writing pure Java code which directly accesses to the JVM for invoking this kind of methods.

Method reification represents another aspect of the reflective method invocation whose performance has to be investigated. In the Java reflection library, it is impossible to reflectively invoke a method before reifying it. Therefore the method reification contributes to the overall performance of the reflective invocation. It is fairly evident that the method reification with our approach cannot take less than the approach provided by the standard reflection library because it requires the loading or the generation on-the-fly of the bytecode for the stub. We have carried out a very simple experiment that involves a class implementing an interface with a single method, in the experiment such a method has been repeatedly reified by using both approaches and by inspecting both the class and the interface. The average of the achieved times is reported in table 2. From this experiment we have discovered that the bytecode loading affects the method reification performance approximately of the 10% whereas the stub generation is more expensive (around the 25% more than the standard approach).

	standard	*on disk*	*on-the-fly*
reification via class inspection	0.005418	0.006055	0.006806
reification via interface inspection	0.005405	0.006561	0.007724
			* *all the reported time are expressed in milliseconds.*

Table 2: **Method reifications in comparison.**

As expected our approach is slower than standard one with respect to the method reification but to verify the usefulness of the mechanism we have to evaluate how long it takes to become advantageous, that is, how many invocations are necessary to amortize method reification overhead and take advantage from the method invocation speed up. To evaluate this aspect we have to consider two factors: i) the time needed to reify the method, and ii) the time necessary to invoke it. The former is an overhead that we have to pay once whereas the latter is paid at each invocation, after these considerations and from the figures reported in tables 1 and 2 it is easy to conclude that to absorb the initial overhead due to the method reification we have to invoke it few times through an interface and around twelve times through an instance. We can consider such an amortize factor as acceptable because the same stub is shared by all the methods (by all the fields and constructors as well) of a class and it is reasonable that, in a real program, we have to reflectively access many times to the attributes (methods, constructors and fields) of the same class.

5 CONCLUSIONS

In this paper we have exposed our idea for optimizing the performances of the reflective Java reflection library focusing our efforts on the reflective method invocation. Basically the idea consists in delegating the method lookup and the late binding to the standard invocation mechanism provided by Java. We have also proved the effectiveness of our solution by implementing it as a library – named *SmartReflection* – that provide the same functionality as the Java core reflection library. In particular, we have presented the replacement for the class `Method` – named `SmartMethod` – which provides method introspection and invocation and the replacement for the class `Class` – named `SmartClass` – which provides the mechanism for transparent method reification. Moreover, we have presented the achieved results and the generator used to unfold the method lookup mechanism and to provide the run-time support to the direct call of each method through the instances of `SmartMethod`.

A different approach to render more efficient the method `invoke()` consists of having a pure object-oriented implementation of the class `Method`. Basically, the class `Method` maintains the same interface but it is an abstract class and its subclasses will embed, in the implementation of their method `invoke()`, the direct call to the method they are reifying. Therefore, instances of `Method` are never created, instances of its subclasses are created and used instead.

Surely a similar approach could be more elegant and flexible than the one proposed in this paper and probably it gives the same benefits or better in terms of performance but it has two major problems that have pressed us to not further consider this approach. First, this approach presupposes to directly instantiate the class reifying the method to invoke rather than instantiate the class `Method` with the right parameters (it does not provide the programmer with a uniform approach to method reification). Second, the first in importance, we have to pay what we gain in terms of performance with an elevate proliferation of classes (a new class for each

method). Besides, we would need a tool which examines the classes and generates all the concrete sub-classes of the class `Method`.

Our approach is still far from being perfect. Most of the time which separates a direct invocation from our reflective invocation is lost in type conversions (we must cast the arguments from `Object` to the type expected by the method), in binding the `SmartMethod` with the stub of the right class (late binding) and in mangling Java code with C++ code through the JNI interface (C++ permits to work around invocation restrictions, e.g., it permits to invoke **`private`** methods). We are studying a way to overcome these flaws and render more and more efficient our reflection library and our invocation mechanism.

ACKNOWLEDGMENTS

This work is based on preliminary results published in [2], the idea that allowed us to get such results has come out from a lively and cleaver discussion about Java reflection with Angelika Langer that the author wishes to thank.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, December 1997.

[2] Walter Cazzola. SmartMethod: an Efficient Replacement for Method. In *Proceedings of the 9th Annual ACM Symposium on Applied Computing (SAC'04)*, pages 1305–1309, Nicosia, Cyprus, on 14th-17th of March 2004. ACM Press.

[3] Walter Cazzola, Shigeru Chiba, and Thomas Ledoux. Reflection and Meta-Level Architectures: State of the Art, and Future Trends. In Jacques Malenfant, Sabine Moisan, and Ana Moreira, editors, *ECOOP'2000 Workshop Reader*, LNCS 1964, pages 1–15. Springer, December 2000.

[4] Shigeru Chiba. A Meta-Object Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 285–299, Austin, Texas, USA, October 1995. ACM.

[5] Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.

[6] Ira R. Forman and Nate B. Forman. *Java Reflection*. Manning Publications, November 2004.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[8] Kevin Hoffman, Jeff Gabriel, Denise Gosnell, Jeff Hasan, Christian Holm, Ed Musters, Jan Narkiewickz, John Schenken, Thiru Thangarathinam, Scott Wylie, and Jonothon Ortiz. *Professional .NET Framework*. Wrox Press., 2001.

[9] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, June 1999.

[10] Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

[11] Hidehiko Masuhara and Akinori Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In Eric Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, LNCS 1445, pages 418–439. Springer-Verlag, July 1998.

[12] Microsoft Corporation. .NET Framework Developer's Guide: Reflection Overview. Technical report, Microsoft Developer Network (MSDN), 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconreflectionoverview.asp.

[13] SUN Microsystems. Java™ Core Reflection API and Specification. Technical report, SUN Microsystems, February 1997.

[14] SUN Microsystems. Java™ Remote Method Invocation - Distributed Computing for Java. White paper, SUN Microsystems, 1998. Internet Publication - http://www.sun.com.

[15] Akinori Yonezawa and Satoshi Matsuoka, editors. *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192. Springer, September 2001.

ABOUT THE AUTHORS

Walter Cazzola (Ph.D.) is currently an assistant professor at the Department of Informatics and Communication (DICo) of the Università degli Studi di Milano, Italy. His research interests include reflection, aspect-oriented programming, programming methodologies and languages. He has written and has served as reviewer of several technical papers about reflection and aspect-oriented programming.