Jumping to the Meta Level

Behavioral Reflection can be fast and flexible

Michael Golm, Jürgen Kleinöder

University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Operating Systems)
Martensstr. 1, D-91058 Erlangen, Germany
{golm, kleinoeder}@informatik.uni-erlangen.de

Abstract. Fully reflective systems have the notion of a control transfer from base-level code to meta-level code in order to change the behavior of the base-level system. There exist various opinions on how the programming model of a meta architecture has to look like. A common necessity of all models and systems is the need to intercept messages and operations, such as the creation of objects. We analyze the trade-offs of various message interception mechanisms for Java. We show their impact on the meta-level programming model and performance. We demonstrate that it is beneficial to integrate the interception mechanism with the virtual machine and the just-in-time compiler.

1 Introduction

An important aspect that determines the applicability and performance of a meta architecture is the way the connection between the base-level code and the meta-level code is established and how control and information flows between the two levels.

The modification of the control flow of base-level programs - called *behavioral reflection* - is a very important part of a fully reflective architecture. Most often the meta-level code is hooked into the message passing mechanism. The process of modifying the message passing mechanism is called *reification of message passing*. The importance of this kind of reflection for "real-live problems" can be observed by looking at the huge amount of design patterns whose sole purpose is the interception of messages. The most obvious examples are the proxy and adapter patterns [10].

Although we focus our attention on Java systems, we also consider systems that are based on languages, such as Smalltalk and C++, provided that the features we are interested in are not language-dependent.

This paper is structured as follows. In Section 2 and 3 we evaluate techniques for message interception. In the remaining sections of the paper we describe a technique that requires modifications of the Java Virtual Machine (JVM). Therefore we describe the relevant structures and mechanisms of the JVM in section 4. Section 5 discusses special problems of information transfer between base and meta level. Section 6 describes the changes to the JVM that where necessary to support reflection in our own reflective Java system - *metaXa*. Section 7 describes experiments that we performed to measure the overhead of behavioral reflection when a JIT compiler is used.

2 Implementation of behavioral reflection

There exist several systems with fairly different approaches to reifying message passing. We evaluate message interception techniques, that are used in these systems, according to the criteria *cost*, *functionality*, and *transparency.*

2.1 Costs

The cost for method reification must be paid as an overhead for a certain mechanism or as overhead for the whole system. The cost is paid somewhere between the time the program is developed and the time the mechanism is used. We take a closer look at the following costs:

Virtual method call. The invocation of a (virtual) method is a very frequently executed operation that determines the performance of the whole system. The cost of such a method call in a reflective system should be the same as in a non-reflective system.

Reflective method call. The cost of a reflective method call will naturally be higher than that of a normal method call. However, it should not be too high and it should scale with the complexity of the meta-level interaction. Short and fast meta-level operations should only produce a small call overhead, while complicated operations, that require a lot of information about the base level, could create a larger overhead.

Installation. Installation costs are paid when a method is made reflective. This cost is paid at compile time, load time, or run time.

Memory. Reflective systems consume additional memory for additional classes, an enlarged class structure, enlarged objects, etc.

2.2 Functionality

The implementation technique for the meta-level jump also affects the functionality of the whole meta architecture. We will examine which functionality each of the implementation techniques is able to deliver, according to the following questions:

- Can behavioral reflection be used for single objects or only for object groups, such as all instances of a class?
- What context information is available at the meta-level method? Is it possible to obtain parameters of the intercepted call? Is it possible to obtain information about local variables of the call site?
- Is it possible to build libraries of reusable metaobjects?

2.3 Transparency

As base-level code and meta-level code should be developed and maintained independently from each other, it is essential that the actions of the meta level are transparent to the base-level program. Programmers should not be forced to follow certain coding rules to exploit the services of the meta level. Forcing the programmer to follow certain coding rules would impede resuse of class libraries and applications - one of the most compelling reasons for extending an *existing* systems. If we extend an existing language or runtime system we should take care to guarantee the downward compatibility of the system.

A change in the behavior of a program should be actuated by the meta-level programmer and should not be a side effect of the implementation of the reflective mechanism.

When we consider transparency in the Java environment we must ensure that:

- the identity of objects is preserved (otherwise the locking mechanism would be broken and data inconsistencies can result),
- inheritance relations between classes are not visibly changed (such a change can become visible if the instanceof operator or an invocation of a superclass method or constructor differ from their normal behavior)

3 Implementation techniques

3.1 The preprocessor approach

With the availability of JavaCC [23] and a Java grammar for this compiler generator it seems to be easy to build Java preprocessors. However, we encountered several difficulties while building a preprocessor that inserts code at the begin and end of a method (reification at the callee site) and at each place where a method is called (caller site reification) [2]. While the caller site reification required only an additional try-finally block enclosing the whole method, insertion of code at the caller site was more difficult. If the method has arguments and an expression is passed as argument, this expression has to be evaluated first. This requires introducing new local variables to which the results of these expressions are bound. This must be done confoming to the Java evaluation order.

The preprocessor approach has the disadvantage that the preprocessor must be changed when the base level language is changed. A system that only operates at the bytecode or virtual-machine level must only be modified if the JVM specification is changed. In the past, the Java language was changed more frequently than the JVM specification.

OpenC++ [4] and OpenJava [5] are examples of preprocessors that insert interceptor methods in the base class.

Preprocessor based systems need full information about the used classes at compile time and can not handle dynamic class loading. This can only be done by paying the installation cost at runtime, as done in the classloader and proxy approach.

3.2 The classloader approach

Generating stubs at compile time means that the stub generator has no information about which stubs are really needed, because it is not known which classes are loaded during a program run. To cope with this problem we can move the stub generation from compile time to load time.

The Java system makes the class loading mechanism available to application programs. This reification of class loading is used by systems like JRes [8] and Dalang [25] to modify classes when they are loaded. The original class is renamed and a new class is generated that has the name of the loaded class but contains only stub code to invoke the metaobject. The rest of the program that uses the original class now gets the created class.

By its very nature a classloader approach can only be used for class-based reflection.

Szyperski [24] describes problems that appear when the wrapped object invokes methods at its self reference. The two presented choices are to invoke the method at the wrapper object or at the wrapped object. In our opinion an invocation at the self reference is only a special case of an incoming/outgoing message. So it would be natural not to bypass the wrapper but invoke these methods also at the wrapper object.

3.3 The proxy approach

To overcome the restriction of being class-based, one needs to modify the object code at the time the object is created or even later in the object's life. Therefore the proxy class must be created and compiled at run time. A separate compiler process must be created. This means that there is a high installation cost that has to be paid at runtime.

Identity transparency in the proxy approach means that the proxy and the original object are indistinguishable. This is only ensured if the proxy is the only object that has a reference the original object.

Inheritance transparency can not be accomplished with the proxy approach, because the proxy must be a subclass of the original class [13] . Being a subclass of the wrapped object has the consequence, that all fields of the original object are present in the proxy thus duplicating the memory cost. These problems can only be avoided, if the original object is exclusively used via an interface [22]. Then the proxy implements the same interface as the original class.

The proxy approach is also used in reflective systems that are based on Smalltalk [11], [19]. Smalltalk has a very weak type system which makes it easily possible to replace an object with a proxy. The object and the proxy need not even provide the same interface. The proxy object only defines the “doesNotUnderstand” method, which is called by the Smalltalk runtime if the called method is not implemented by the receiver. Smalltalk systems implement the method invocation mechanism in a way that adds only little overhead when using the “doesNotUnderstand” method.

3.4 The VM approach

Being able to extend the JVM provides complete functionality while preserving transparency.

There are two places where a method call can be intercepted: at the caller or at the callee. At first sight this seems to be no difference at all. But when considering virtual methods one immediately sees the difference. If we want to intercept all methods that are invoked at a specific object X of class C, we have to intercept the method at the callee. Otherwise we are forced to modify every caller site that invokes a method at class C or its superclasses. On the other side, if we are interested in the invocations that are performed by a specific object we have to modify the caller site.

When extending a JVM one has several alternatives to implement the interception mechanism:

New bytecode. All interesting bytecodes (invokevirtual and invokespecial) are replaced by reifying pedants (e.g. invokevirtual_reflective). This is a clean way to create reflective objects but requires an extension of the bytecode set. Replacing the original bytecode by a new one has only a small installation cost. This new bytecode must be understood by the interpreter and the JIT compiler. Additionally, if the system allows the structural reification of a method's bytecodes the new bytecode should not become visible, because this bytecode is only an implementation detail of the message interception mechanism.

Extended bytecode semantics. The interpretation process of interesting bytecodes is extended. Guaraná [21] uses this approach. Creating additional cost for each virtual method invocation degrades the performance of the whole system. Using the kaffee JIT compiler, Oliva et al. [21] measured the invokevirtual overhead on different Pentium and Sparc processors .The overhead they measured on the Pentium Pro (158%) conforms with our measurement on the Pentium 2 (see Section 7.2 and 7.3).

Code injection. Method bytecodes of C.m() are replaced by a stub code which jumps to the meta space. This approach is compatible to JIT compilation. Because normal Java bytecode is inserted, the code can be compiled and optimized by a normal JIT compiler.

Code modification is a powerful technique, that can be used in a variety of ways.

- *Kind of injected code*: The usual way is to execute a virtual method, but we can imagine a static method or just code fragments. Static methods and code fragments can be inlined in the base-level code.
- *Granularity and Location*: Aspect Oriented Programming [14] is such a powerful approach because it provides very fine-graned code injection. This, however, requires to write a complex injection mechanism - called weaver- that can not be generally reused. Furthermore the combination of different aspects, i.e. different code fragments, leads to complex interferences between the injected code sequences.

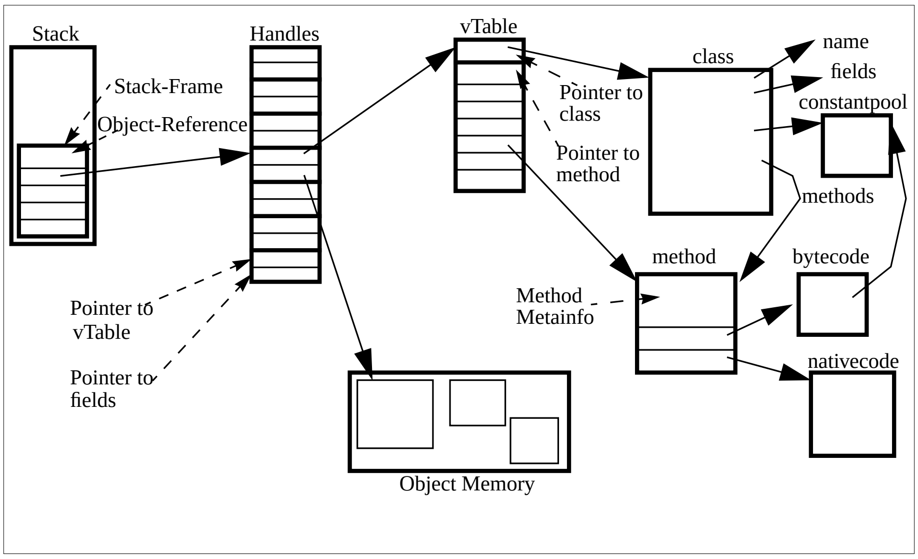

Fig. 1 Classes, Methods, and References in a JVM

Most meta architectures allow the injection of code only at distinct injection points, such as method calls. This makes the job of code injection a lot more easier.

- *Kind of information for the injected code*: This depends on the location of the injected code. Before a method call, the method parameters and local variables of the caller could be interesting; before a new operation, the class of the object is interesting.

4 Java VM implementations

To understand the modifications of the virtual machine that are necessary to enable reification of message passing, one needs a overall understanding of the data structures and mechanisms used inside a JVM. While the JVM specification [16] does not enforce a specific implementation we will base our discussion on the JVM structure outlined in [26]. There exist many JVM implementations that differ from this structure in several aspects [20], [6], [7].

Fig. 1 sketches the relationship between the stack, references, objects, classes, and methods. Local variables are stored in a stack frame. If the local variable has a reference type it contains a pointer to a handle. This handle contains a pointer to the object data and to the virtual method table (vTable). The vTable contains a pointer to the object's class and a list of pointers to the object's methods.

When a virtual method is invoked, the callee reference and all method arguments are pushed on the stack. Then the invokevirtual bytecode is executed. The invokevirtual opcode has three arguments: a class name, a method name and a method signature that are all looked up in the constantpool of the class. The constantpool entry must be resolved to a pointer to a class and an index into the vTable of this class. The method is then looked up using the vTable of the receiver object.

5 Passing information to the meta level

The cost of the meta-level jump itself is proportional to the information that is passed to the meta level. Different metaobjects have different requirements on the kind of information and the way this information is passed. In the following we assume that meta-level code is represented by a virtual method.

Meta interaction is cheapest if the meta level only needs a trigger, just informing the metaobject about the method call but not passing any information to the metaobject.

A bit more expensive is passing control to a meta-level method that has the same signature as the intercepted base-level call [12]. This interception mechanism has the advantage that it is not necessary to copy arguments.

An interaction that is more useful and is commonly used by many meta-level programs in metaXa is the invocation of a generic meta-level method. All arguments of the base-level call are copied into a generic container and this container is passed to the metaobject. The metaobject can then inspect or modify the arguments or pass them through to the original base-level method.

Some sophisticated metaobjects also need information about the broader context where a method was called. Especially the call frame information is needed in addition to the arguments. This information is also packed into a generic container and passed to the metaobject.

While the simple passing strategies are appropriate for situations where the interaction with the meta level has to be fast, some metaobjects need more complex strategies to fulfill their tasks. Thus, we obviously need a mechanism to configure the kind of interaction. Such a configuration can be regarded as a meta-meta level. This meta-meta level is concerned with the kind of code that is provided by the meta level and how this code is injected into the base-level computation. It is also concerned with the way information is passed to the meta level and the kind of this information. The JIT compiler that is described in Section 7 can be considered as such a meta-meta level.

6 How it is done in metaXa

In this section we describe the modifications to the virtual machine necessary to build a system that allows reification of message passing. The modifications allow a level of *transparency* and *functionality* that can not be achieved with the other approaches. In Section 7 we explain how the *cost* can be minimized by using a JIT compiler.

6.1 Shadow classes

Class-based meta architectures associate one metaobject with one base-level class. The reference to the metaobject can be stored in the class structure. Instance-based meta architectures can associate a metaobject with a specific base-level object. The metaobject reference must be stored in an object-specific location. One could store it together with the object fields or in the object handle. Storing it in the handle has the advantage to be reference-specific. Another interesting idea is to generate the slot only when needed, as in MetaBeta with dynamic slots [3]. We have not investigated yet if this mechanism can be implemented efficiently in a JVM. Storing the metaobject reference in the handle means an additional memory consumption in the handle space of 50%. This also applies for VMs where references point directly to the object, without going through a handle. In such an architecture the size of each object must be increased by one word to store the metaobject link.
In metaXa we use a different mechanism to associate the meta object with the base-level object. When a metaobject is attached to an object, a *shadow class* is created and the class link of the object is redirected to this shadow class. The shadow class contains a reference to the associated meta object.

A metaobject can be used to control a number of base-level objects of the same class in a similar manner. It is not necessary to create a shadow class for every object. Shadow classes are thus installed in two steps. In the first step, a shadow class is created. In the second step, the shadow class is installed as the class of the object. Once a shadow class is created, it can be used for several objects.

A shadow class C' is an exact copy of class C with the following properties:

- C and C' are undistinguishable at the base level
- C' is identical to C except for modifications done by a meta-level program
- Static fields and methods are shared between C and C'.

The base-level system can not differentiate between a class and its shadow classes. This makes the shadowing transparent to the base system. Fig. 2 shows, how the object-class relation is modified when a shadow class is created. Objects A and B are instances of class C. A shadow class C' of C is created and installed as class of object B. Base-level applications can not differentiate between the type of A and B, because the type link points to the same class. However, A and B may behave differently. Several problems had to be solved to ensure the transparency of this modification.

Class data consistency. The consistency between C and C' must be maintained. All class-related non-constant data must be shared between C and C', because shadow classes are targeted at changing object-related, not class-related properties. Our virtual machine solves this problem by sharing class data (static fields and methods) between shadow classes and the original class.

Class identity. Whenever the virtual machine compares classes, it uses the original class pointed to by the *type* link (Fig. 2). This class is used for all type checks, for example

- checking assignment compatibility

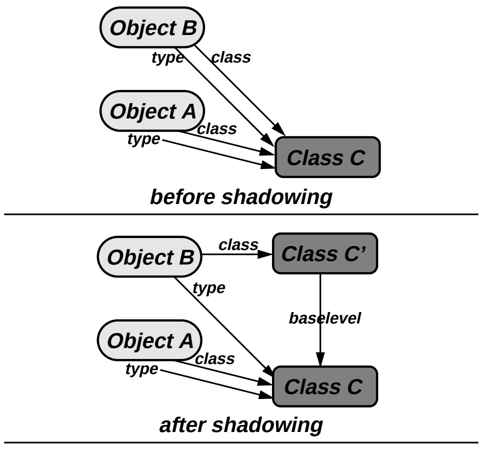

Fig. 2 The creation of a shadow class

- checking if a reference can be cast to another class or interface (checkcast opcode)
- protection checks

Class objects. In Java, classes are first-class objects. Testing the class objects of C and C' for identity must yield true. In the metaXa virtual machine every class structure contains a pointer to the Java object that represents this class. These objects are different for C and C', because the class objects must contain a link back to the class structure. Thus, when comparing objects of type class, the virtual machine actually compares the type links.

Because classes are first class objects it is possible to use them as mutual exclusion lock. This happens when the monitorenter/monitorexit bytecodes are executed or a synchronized static method of the class is called. We stated above that shadowing must be transparent to the base level. Therefore the locks of C and C' must be identical. So, metaXa uses the class object of the class structure pointed to by the type link for locking.

Garbage Collection. Shadow classes must be garbage collected if they are no longer used, i.e. when the metaobject is detached. The garbage collector follows the *baselevel* link (Fig. 2) to mark classes in the tower of shadow classes (tower of metaobjects). Shadowed superclasses are marked as usual following the superclass link in the shadow class.

Code consistency. Some thread may execute in a base-level object of class C when shadowing and modification of the shadow takes place. The system then guarantees that the old code is kept and used as long as the method is executing. If the method returns and is called the next time, the new code is used. This guarantee can be given, because during execution of a method all necessary information, such as the current constantpool or a pointer to the method structure, are kept on the Java stack as part of the execution environment.

Memory consumption. A shadow class C' is a shallow copy of C. Only method blocks are deep copied. A shallow class has a size of 80 bytes. A method has a size of 52 bytes. An entry in the methodtable is a pointer to a method and has a size of 4 bytes. Hence, the cost of shadowing is a memory consumption of about 400 bytes for a shadow class with five methods. The bytecodes of a method are shared with the original class until new bytecodes are installed for this method.

Inheritance. During shadow class creation a shadow of the superclass is created recursively. When the object refers to its superclass with a call to super the shadowed superclass is used. All shadowed classes in the inheritance path use the same metaobject. Fig. 3 shows what happens if a metaobject is attached to a class

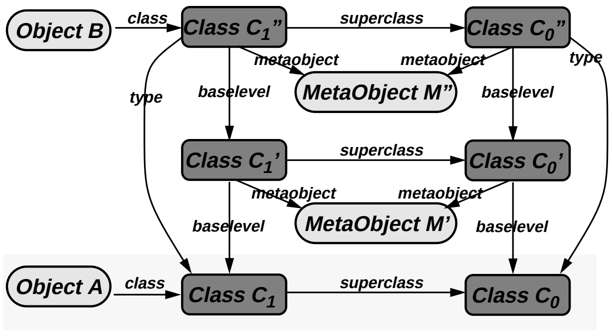

Fig. 3 Metaobjects and inheritance

that has several superclasses. The shaded area marks the original configuration. Object A and B are of class C_1 which is a subclass of C_0. When the metaobject M' is attached to object B, a shadow class C_1' of C_1 is created. Later, the metaobject M" is attached to B. This causes the creation of shadow class C_1“.

Original behavior. In addition to the metaobject link a shadow class in metaXa also needs a link baselevel to the original class. It is used when resorting to the original behavior of the class. In all non-shadow classes the baselevel link is null. The base-level link is used to delegate work to the original class of the object.

6.2 Attaching metaobjects to references

In metaXa it is also possible to associate a meta object with a reference to an object. If a metaobject is attached to a reference, method invocations via this *reference* must be intercepted. A reference is a pointer, which is stored on the stack or in objects and points to a handle (Fig. 1). If the reference is copied - because it is passed as a method parameter or written into an object - only the pointer to the handle is copied, still pointing to the same handle as before. To make a reference behave differently, the handle is copied and the class pointer in the handle is changed to point to a shadow class. After copying the handle it is no longer sufficient to compare handles when checking the identity of objects. Instead, the data pointers of the handles are compared. This requires that data pointers are unique, i.e. that every object has at least a size of one byte. The metaXa object store guarantees this. Fig. 4 shows the handle-class relation after a shadow class

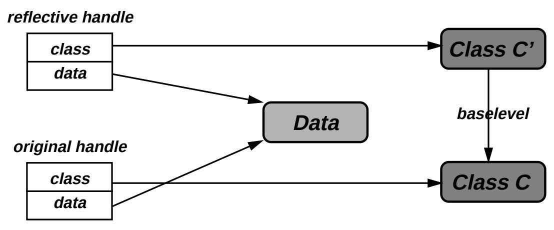

Fig. 4 Attaching to a reference

was installed at a reference. A reference is represented by a handle, which contains a link to the object's data and a link to the object's type (class or method-table). If a shadow class must be attached to the handle (original handle), the handle is cloned (reflective handle) and the class link is set to the shadow class.

6.3 Creating jump code using a bytecode generator

We will now explain how the code to jump to the metaobject is created using a bytecode generator. The problems when directly generating native code are similar. When inserting code into an existing method the jump targets and the exception tables must be corrected. This is done automatically by our very flexible bytecode generator. This generator first translates a method's bytecodes to objects. One can then replace and modify these bytecode objects, and insert new ones. This flexible generator allowed us to easily implement different stub generators but it also leads to high installation costs.

We describe three different mechanisms for which stubs are created:

- a method is sent by the base-level object (*outcall*),
- an object is created by the base-level object (*new*),
- a method is received by the base-level object (*incall*).

outcall. An outgoing messages is caused by the invokevirtual opcode. The other opcodes that invoke methods are invokespecial to call constructors or superclass methods and invokestatic to call class methods. They are not interesting for our current considerations. The invokevirtual opcode has three arguments: a class name, a method name, and a method signature, that are all looked up in the constantpool of the class. The code generator creates the following code and replaces the invokevirtual opcode by a code sequence that performs the following operations:

(1) Create an object that contains information about the method call. This object contains the class name, method name, and signature of the called method. It also contains the arguments of the method call.
(2) At the metaobject invoke a method that is responsible to handle this interception.

new. The new bytecode is replaced by a stub that looks similar to the outcall stub.

incall. While it is rather obvious how the interception code has to look like in the outcall and new interceptions, there are alternatives for the interception of incoming invocations.

- One solution would be to insert calls to the meta level at the beginning of the method. But then it is not possible to avoid the execution of the original method other than throwing an exception. Furthermore, the return from the method can not be controlled by simply replacing the return bytecode. Instead, all exceptions have to be caught by inserting a new entry into the exception table.
- A simpler way of executing code before and after the original method is to completely replace the original method by the stub code that invokes the metaobject. The metaobject then is free to call the original method. This call executes the bytecodes of the original method by doing a lookup in the original class.

7 Integration into the JIT compiler

A central part of metaXa is the bytecode rewriting facility that was explained in the previous section. After we have implemented a just-in-time compiler (JIT) for the metaXa VM [15] it seems more appropriate to directly create instrumented native code instead of instrumented bytecode. Such an integration has several advantages:

- The installation cost is contained in and dominated by the JIT compilation cost, which must be paid even if no meta interaction is required.
- The JIT can inline a call to a meta object, which makes the interaction with meta objects very fast.

Another project that also uses a JIT for meta-level programming is OpenJIT [18]. OpenJIT is a JIT compiler that is designed to be configurable and extensible by metaprograms.

There are other systems that use partial evaluation to remove the overhead of the tower of meta objects, e.g. ABCL/R3 [17]. Because we use the OpenC++ [4] model of method interception, rather than the more general architecture of a meta-circular interpreter, we are faced with a different set of problems.

7.1 Inlining of meta-level methods

The overhead of a meta interaction should grow with the amount of work that has to be done at the meta level. This means especially that simple meta-level operations, such as updating a method invocation counter, should execute with only a minimal overhead. On the other side, it should be possible to perform more complex operations, such as forwarding the method call to a remote machine. In this case a user accepts a larger overhead, because the cost of the meta-level jump is not relevant compared to the cost of the meta-level operation, e.g. a network communication.

The reduce the meta-jump overhead in case of a small meta-level method we use method inlining. Inlining methods results in speedup, because method-call overhead is eliminated and inter-method optimizations can be applied. These optimizations could perform a better register allocation and elimination of unnecessary null-reference tests. A method can be inlined as long as the code that has to be executed is known at compile time. Therefore, one can generally inline only non-virtual (private, static, or final) methods. Fortunately, it is also possible to inline a method call, if the method receiver is known at (JIT-) compile time. When inlining meta-level methods, the metaobject usually already exists and is known to the JIT compiler.

The principle of inlining is very simple. The code of the method call (e.g. invokestatic) is replaced by the method code. The stack frame of the inlined method must be merged into the stack frame of the caller method. The following example illustrates this merging. If a method test() contains the method call a = addInt(a, 2), this call is compiled to

```
aload 0
iload 1
iconst_2
invokenonvirtual addInt(II)I
istore 1
```

The method int addInt(int x, int y) { return x+y; } is compiled to

```
iload 1
iload 2
iadd
ret
```

The merged stack frame has the following layout:

merged stack	stack of test()	stack of addInt()
0	local 0	
1	local 1	
2	operand 0	local 0
3	operand 1	local 1
4	operand 2	local 2
5		operand 0
6		operand 1

The created code would look as follows (bytecodes have stack positions as operands):

```
aload 0 -> 2
iload 1 -> 3
iconst #2 -> 4
checkreference 2
iload 3 -> 5
iload 4-> 6
iadd 5, 6 -> 5                  addInt()
istore 5 -> 2
istore 2 -> 1
```

Several optimizations can now be applied to the code. Provided that the method test() only contains the addInt() method call, the code can be transformed using copy propagation and dead code elimination [1] to

```
checkreference 0
iadd 1, #2 -> 5
istore 5 -> 2
istore 5 -> 1
```

Virtual methods can be inlined, if no subclasses override the method. If the JVM loads a subclass that overrides the method, the inlining must be undone. Such code modifications must be done very carefully, because a different thread could be executing this code.

7.2 Experiments

We measured the execution times for the following operations:

ccall and c++call. To have an idea how good or bad the performance of the meta-level interaction actually is, we measured the time for a function call in C and for a virtual method call in C++ (using the gcc compiler with -O2 optimization).

vcall. Invocation of a virtual method through a method table, doing the required null reference check before the method lookup. The called method immediately returns. x instructions are executed on the code path. x memory locations are read.

intercept. A call to a virtual method of a different object is inserted before the original method invocation. The object to be called is known at compile time. This allows inlining of the object reference into the code. It is necessary to register this reference with the garbage collector and unregister the reference if the compiled method is garbage collected. The injected method is called *meta method* in the following. The meta method has the same signature as the original method. The meta method returns immediately.

counter. A very common application for meta methods is counting of base-level method invocations. The meta method increments the counter variable. If this counter reaches a certain value (e.g. 5000), a different method (e.g. action()) is called to perform a meta-level computation, e.g. to make a strategic decision. The bytecode for this meta method is shown below:

```
Method void count()
  0 aload_0
  1 getfield #8 <Field int counter>
  4 sipush 5000
  7 if_icmple 14
 10 aload_0
 11 invokevirtual #7 <Method void action()>
 14 return
```

We measured the times for the following operations:

- not taken: The threshold is never reached and the action method is never called.
- taken1: The threshold is set to 1. This means that the action method is invoked before each invocation of the base-level method.
- taken10 and taken100: The threshold is set to 10 (100). The action method is invoked before every 10th (100th) call of the base-level method.

argobj. The method arguments are copied into a parameter container. This container is then passed to the meta method. The elements of the container are very special because they must contain reference types as well as primitive types, which is not possible when a Java array is used as the container. We rejected the solution to use wrapper classes for the primitive types (the Reflection API does it this way), because this means that additional objects must be created for the primitive parameters. Instead we use a container whose element types can be primitive types and reference types at the same time. As this is not allowed by the Java type system, we have to use native methods to access the container elements. Furthermore, the garbage collector must be changed to cope with this kind of container objects. The test performs the following operations:

- the parameters are read from the stack and written into the container object
- a virtual method is called with this object as parameter
- this method uses a native method to invoke the original method with the original parameters

This test is an optimized version of the metaXa method interception.

inlinedobj. We now instruct the JIT to inline the call. Because the meta method passes the call through to the original method, the result of the inlining should be the original method call.
A requirement for inlining is, that the implementation, that is used to execute the method call, is known at JIT-compile time. Therefore the meta method must either be static, final, or private or the class of the metaobject is known.
The JIT has some further difficulties inlining the method, because the parameter container is accessed using native calls and native methods cannot generally be inlined. So we either have to avoid native methods or enable the JIT to inline them. In the discussion of the argobj test we explained why avoiding the native methods is not possible. In order to inline the native method, the JIT now queries the VM for a native code fragment that can be used as the inlined method. In most

cases the VM will not provide such a code segment. For the native methods that access the parameter container such a native code equivalent is provided.
To eliminate the object allocation the JIT must deduce that the parameter container object is only used in this method so that the object can be created on the stack. Our JIT does not do such clever reasoning but it can be implemented using the algorithms described in [9]. In our current implementation we give the JIT a hint that the object is used only locally. Using its copy propagator the JIT can now remove the unnecessary parameter passing through the parameter container.

7.3 Performance results

We were interested in the performance of each of the discussed code injection techniques when using a native code compiler. We did not measure the installation cost (which would include the compilation time) because our JIT compiler is not optimized yet. It operates rather slowly and generates unoptimized code. We used a 350 MHz Pentium 2 with 128 MB main memory and 512 kByte L2 cache. The operating system is Linux 2.0.34. The system was idle.

We first measured the time for an empty loop of 1,000,000 iterations. Then we inserted the operation, that we were interested in, into the loop body and measured again. The difference between the empty loop and the loop that executes the operation is the time needed for 1,000,000 operations. Times for an empty loop of 1,000,000 iterations in C and compiled Java code are 15 milliseconds.

Operation	**execution time (in nanoseconds)**
ccall	17
c++call	35
vcall	65
vcallchecked	164
intercept	190
counter/not taken	145
counter/taken1	223
counter/taken10	195
counter/taken100	151
argobj	14000
inlinedobj	250

7.4 Analysis

The argobj operation is so extremely slow compared to the other operations, because it is the only operation that jumps into the JVM, executes JVM code, and allocates a new object. All other operations execute only very few JIT-generated machine instructions. However, a general pupose interception mechanism must allow the interceptor to access the method parameters in a uniform way and independent from the method signature. So there seems to be no alternative to using an parameter container. To improve the performance of the interception the compiler must try to eliminate the object allocation overhead, which is possible - as the inlinedobj test shows. We expect that the performance of this operation can be further improved when our JIT generates code that is more optimized.

8 Summary

We have studied several approaches to implement reification of message passing. It became apparent that only a system that integrates the meta architecture with the JVM and the JIT compiler delivers the necessary performance, functionality, and transparency to make behavioral reflection usable. The method interception mechanism can aggressively be optimized by certain techniques, such as inlining of methods and inlining of object allocations. While the JIT compiler can automatically perform such micro-level optimizations it is necessary to have a meta-meta level that controls the configuration of the meta level and selects the appropriate mechanism for the interaction between base level and meta level. For optimal performance this meta-meta level should be part of the just-in-time compiler, or should at least be able to influence the code generation process.

9 References

1. V. A. Aho, R. Sethi, D. J. Ullman: *Compilers: Principles, Techniques, and Tools* , Addison-Wesley 1985
2. M. Bickel: *Realisierung eines Prototyps für die Koordinierungs-Metaarchitektur SMArT in Java. Studienarbeit IMMD 4*, March 1999
3. S. Brandt, R. W. Schmidt: The Design of a Metalevel Architecture for the BETA Language. In *Advances in Object-Oriented Metalevel Architectures and Reflection* , CRC Press, Boca Raton, Florida 1996, pp. 153-179
4. S. Chiba, T. Masuda: Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of the European Conference on Object-Oriented Programming '93,* Lecture Notes in Computer Science 707, Springer-Verlag 1993, pp. 482–501
5. S. Chiba, M. Tatsubori: Yet Another java.lang.Class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems,* 1998
6. T. Cramer: Compiling Java Just-in-time. In *IEEE Micro* , May 1997
7. R. Crelier: *Interview about the Borland JBuilder JIT Compiler.*
8. G. Czajkowski, T. von Eicken: JRes: A Resource Accounting Interface for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, ACM Press 1998, pp. 21-35
9. J. Dolby, A. A. Chien: An Evaluation of Object Inline Allocation Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, ACM Press 1998, pp. 1-20

10. E. Gamma, R. Helm, Johnson R., J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* , Addison-Wesley, Reading, MA 1994
11. B. Garbinato, R. Guerraoui, K. Mazouni: Implementation of the GARF Replicated Objects Platform. In *Distributed Systems Engineering Journal* , March 1995
12. J. d. O. Guimaraes: Reflection for Statically Typed Languages. In *Proceedings of the European Conference on Object-Oriented Programming '98,* 1998, pp. 440-461
13. U. Hölzle: Integrating Independently-Developed Components in Object-Oriented Languages. In *Proceedings of the European Conference on Object-Oriented Programming '93,* Lecture Notes in Computer Science 512, July 1993
14. G. Kiczales, J. Lamping, C. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming '97,* Lecture Notes in Computer Science 1357, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo 1997
15. H. Kopp: *Design und Implementierung eines maschinenunabhängigen Just-in-time-Compilers für Java. Diplomarbeit (Masters Thesis) IMMD 4,* Sep. 1998
16. T. Lindholm, F. Yellin: *The Java Virtual Machine Specification.* Addison Wesley 1996
17. H. Masuhara, A. Yonezawa: Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of the European Conference on Object-Oriented Programming '98,* 1998, pp. 418-439
18. S. Matsuoka: OpenJIT - A Reflective Java JIT Compiler. In *OOPSLA '98 Workshop on Reflective Programming in C++ and Java,* UTCCP Report, Center for Computational Physics, University of Tsukuba, Tsukuba, Oct. 1998
19. J. McAffer: Engineering the meta-level. In *Proceedings of Reflection '96,* 1996
20. Microsoft: *Microsoft Web Page. http://www.microsoft.com/java/sdk/20/vm/Jit_Compiler.htm* 1998
21. A. Oliva, L. E. Buzato: The Design and Implementation of Guaraná. In *COOTS '99,* 1999
22. T. Riechmann, F. J. Hauck, J. Kleinöder: Transitiver Schutz in Java durch Sicherheitsmetaobjekte. In *JIT - Java Informations-Tage 1998,* Nov. 1998
23. SunTest: *JavaCC - A Java Parser Generator,* 1997
24. Z. Szyperski: *Component Software - Beyound Object-Oriented Programming,* ACM Press 1998
25. I. Welch, R. Stroud: Dalang - A Reflective Java Extension. In *OOPSLA '98 Workshop on Reflective Programming in C++ and Java,* UTCCP Report, Center for Computational Physics, University of Tsukuba, Tsukuba, Oct. 1998
26. F. Yellin, T. Lindholm: Java Internals. In *JavaOne '97* 1997