The Implementation of Guaraná on Java$^{\mathrm{TM}}$

Alexandre Oliva
Luiz Eduardo Buzato

Relatório Técnico IC–98-32

Setembro de 1998

The Implementation of **Guaraná** on Java^{TM}

Alexandre Oliva
oliva@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

September 1998

Abstract

Guaraná is a reflective architecture that aims at simplicity, flexibility, security and reuse of meta-level code. It is implemented as an extension of *Kaffe OpenVM*^{TM}, a free implementation of the Java^{TM} Virtual Machine.

We describe the Java classes that implement the meta-object protocol of **Guaraná**, and the modifications introduced in the virtual machine to intercept and reify of operations.

Finally, we evaluate the performance impact of our modifications, and suggest some optimizations that may be implemented in the future.

1 Introduction

Guaraná [3]is a software architecture for computational reflection [2, 4] that introduces a runtime meta-object protocol that allows for dynamic composition of meta-objects in order to build up potentially complex meta-level behavior. This leads to the development of simple, coherent meta-objects. The possibility for their composition improves code reuse. The fact that this composition can be established and modified at run time, on a per-object basis, makes **Guaraná** flexible. Finally, its meta-object protocol provides controlled access to the meta-level of an object, helping separate meta-level from base-level concerns, just like encapsulation helps with the separation of interface from implementation in the object-oriented paradigm.

In the next section, we briefly describe the reflective architecture of **Guaraná**. Section 3 contains a description of our implementation of this architecture, extending a freely-available Java[1] Virtual Machine. In Section 4, we present some figures about the impact of **Guaraná** on the performance of applications, then we list some possible future optimizations in Section 5. Finally, in Section 6, we summarize the main points of the text.

[1]Java is a trademark of Sun Microsystems, Inc.

2 Reflective Architecture

This section assumes familiarity with concepts pertinent to computational reflection. Although not strictly required, some familiarity with **Guaraná** is desirable, as this section serves only to introduce **Guaraná** in a very concise manner. For a detailed explanation, please refer to [3].

Any object may be directly associated with at most one meta-object, called its primary meta-object. If such an association exists, the kernel of **Guaraná** will intercept and reify all method invocations and field accesses made to the base-level object and present them to the associated primary meta-object. This meta-object may reply with a result for the operation, a replacement operation—to be performed instead of the requested one—, or a request to observe or modify the result of the operation (or its replacement), after it is performed.

One of the actions that a meta-object may take when dealing with an intercepted operation or result is to delegate it to other meta-objects. When a meta-object plays the role of delegator, it is called a composer. Composers have a central role in the reflective architecture of **Guaraná**: they are fundamental to allow the combination of several autonomous meta-objects into a meta-configuration. As composers are themselves meta-objects, they may be composed further, in a potentially infinite hierarchy. Furthermore, every meta-object, just like any other object, can have its own meta-configuration, in another potentially infinite orthogonal hierarchy.

There is no way for an object to find out what meta-object is its primary meta-object. This impossibility is intentional: we believe that an object should not interact with its meta-objects; it should not even be aware of their existence, otherwise the separation of concerns between the levels would be broken. The meta level, on the other hand, is responsible for controlling the base level, so a meta-object does know what objects are associated with it.

Meta-objects have privileged access to objects whose meta-configurations they belong to. They can create operations that violate standard access control by using operation factories, distributed whenever a meta-object is associated with the base-level object, and invalidated as soon as the primary meta-object is replaced.

The composition structure determines a hierarchy of authority and control over the base-level object. The primary meta-object is the highest authority, so it is able to restrict the ability of their components to create operations or set their results. The identity of the primary meta-object of an object is protected, so as to prevent this hierarchy from being subverted. Thus, a meta-object is usually unable to tell whether it is the primary meta-object of an object, since it is possible for a composer to hide from its components, by interacting with other meta-objects just like the kernel of **Guaraná** would.

Reconfiguration requests are also subject to the approval of the existing meta-configuration. It is possible to request for the replacement of subsets of the meta-configuration of an object, and it is up to the existing meta-configuration to determine the new meta-configuration, that may be any combination of the requested meta-configuration with the existing one.

The meta-configuration of a class can affect the meta-configuration of its instances in two situations: (i) when a reconfiguration request is issued to an object that lacks a meta-configuration, the meta-configuration of its class and of its base classes will be given the opportunity to inspect and modify the requested meta-configuration, and (ii) when an object is instantiated, the meta-configuration of the creator is requested to provide a meta-configuration for the new object, then the meta-configuration of the class of the new object is given the opportunity to try to reconfigure it.

These two kinds of interactions with the meta-configuration of an object use a general mechanism: broadcasting a message to all components of the meta-configuration of an object. This allows for communication between meta-objects without requiring explicit mutual references, and allows the inter-meta-object protocol to be extended without modifications to the interface of class MetaObject.

3 Implementation

Most of **Guaraná** was coded in Java, but some modifications in the Java Virtual Machine were needed, in order to provide for interception of operations such as invocation of methods, read/write from/to fields and array elements, creation of objects and arrays, entries and exits from monitors.

In the next section, we describe the classes that implement the reflective architecture of **Guaraná**. Then, we list the modifications we have introduced in *Kaffe OpenVM*, a free[2] implementation of the JVM developed at Transvirtual Technologies, so that it supports our reflective architecture.

3.1 Classes and Interfaces

All the classes used in the implementation of **Guaraná** are defined in the package `BR.unicamp.Guarana`. In this section, a short description of each class used in the implementation of **Guaraná** is provided.

3.1.1 Guarana

Class Guarana represents the kernel of **Guaraná**. It provides methods for modifying meta-configurations of objects (reconfigure), broadcasting Messages to components of a meta-configuration (broadcast), creating proxy objects (makeProxy) and performing operations (perform). This last method is invoked as part of the interception mechanism: it runs the operation handling protocol, i.e., it presents the reified operation to the primary meta-object of the target object, delivers the operation to the target object, then, if requested, presents the result to the primary meta-object.

Internally, some `private native` methods are used for obtaining and modifying the primary meta-object of an object, and for delivering an operation to its target object.

Some standard Object methods would be useful for gathering information about base-level objects from the meta-level. However, if meta-objects would invoke them directly, these invocations would be intercepted, but then the meta-object might decide to invoke

[2]The *Kaffe OpenVM* is distributed under the terms of the GNU General Public License.

the method again, and infinite recursion would result. Thus, the kernel of **Guaraná** offers alternate implementations of some of these methods, such as toString, hashCode and getClass. They take a reference to a base-level object and produce the result that would be produced by the invocation of the standard implementations of these methods, but they do not interact with the base-level object, removing the risk of infinite recursion. A method getClassName, that obtains the name of a given class, is also provided.

3.1.2 Operation

Every base-level operation addressed to an object or class whose meta-configuration is non-null is reified as an Operation object. This class is mostly implemented in native code, for performance reasons: it would be too expensive to wrap every non-object argument type into an array of arguments.

This class provides methods for querying the operation about its nature (i.e., method or constructor invocation, field read or write, monitor enter or exit, array length read or array element read or write) and arguments (i.e., the target object, the Thread that created the Operation, the method or constructor to be invoked, or the field or array index to be read or written, and the invocation arguments or the value to be written).

There are methods for validating and performing operations. The former checks whether an operation is consistent and can be performed in the base level (because it is possible to create inconsistent operation objects); the latter dispatches the operation for interception, validation and execution.

This class also provides operations for checking whether an operation is a replacement operation and what operations are directly or indirectly replaced by it.

An implementation of method toString, that fully describes the operation, is also provided.

3.1.3 Result

A Result object is implicitly created by the kernel of **Guaraná** after an Operation is executed. It provides methods for querying what Operation it refers to, as well as what is the actual value of the result.

However, results can also be created by MetaObjects to modify the result of an operation, by invoking the `static` methods return**T** (where **T** is Object or a primitive type name) or throwObject. If the kernel of **Guaraná** receives a Result of the throwObject variant as the result of an Operation, it throws or rethrows the contained Throwable, instead of returning control to the caller.

Result objects serve yet another purpose: they can be used by MetaObjects to replace operations, and to request permission to observe or modify the result of an operation after it has been performed. This kind of Result will usually be created and returned by a MetaObject as it handles an Operation.

This class is also implemented mostly in native code, in order to avoid as much as possible wrapping primitive types.

3.1.4 MetaException

Whenever the interception of a base-level operation terminates by throwing an exception, the kernel of **Guaraná** creates a MetaException to encapsulate it. Since MetaException is a subclass of RunTimeException, it can propagate through methods that have not declared it.

3.1.5 MetaObject

The class MetaObject is the root of the class hierarchy for every possible implementation of meta-object. Methods for initialization (initialize) and termination (release), interception of Operations, Results and Messages (handle), reconfiguration (reconfigure) and configuration of new objects (configure) can be overridden by its subclasses.

Although MetaObject is an abstract class, because its instances would not do anything useful, there are no abstract methods. All methods implement reasonable defaults, so that subclasses can focus on relevant methods only.

3.1.6 Composer

A Composer is a meta-object that delegates operations, results and messages to other meta-objects. Thus, it `extends` class MetaObject by adding two other methods, used to query the Composer about which MetaObjects it delegates to. Subclasses may specialize method getMetaObjectsArray so that it returns an array containing all MetaObjects it may delegate to.

A very inefficient implementation of the previous method is provided, based on the Java Enumeration returned by the `abstract` method getMetaObjects. Subclasses *must* implement this method, so that the returned Enumeration iterates on all MetaObjects it may delegate to. However, since an Enumeration may contain arbitrary Objects, some of them may be used to provide additional control information about how and when the Composer delegates to particular MetaObjects. However, there is no standardized form of control information.

3.1.7 SequentialComposer

This is a simple specialization of Composer that maintains an array of MetaObjects, and delegates Operations and Messages (see below) to them sequentially. Results are also delegated sequentially, but in the reverse order.

Besides implementing the standard MetaObject and Composer methods, SequentialComposer provides `static` methods for delegating operations and results to subsets of a meta-object array, as well as for configuring new objects.

3.1.8 Message

This is a Java `interface` that provides no methods at all. Only instances of classes that implement this `interface` can be used as arguments to method broadcast of the kernel of **Guaraná**, to exchange information with components of the meta-configuration of an Object.

This restriction was imposed to prevent arbitrary Objects from being broadcast to meta-configurations. By requiring classes to be defined for new types of message, we avoid possible ambiguities that might have arisen if implementors of different MetaObject had adopted different meanings for predefined classes.

3.1.9 NewObject

This is an implementation of Message that stores a reference to an Object. An instance of this class is created by the kernel of **Guaraná** as part of the Object creation process, if the instantiated class has a `null` meta-configuration.

After an Object is allocated and the meta-configuration of its creator determines its meta-configuration, but before the constructor of the object is invoked, a NewObject Message is broadcast to the meta-configuration of the class of the new Object, so that the meta-configuration of the class can try to affect the meta-configuration of its instances.

The class NewObject implements a single method, that returns a reference to the newly-created Object.

3.1.10 NewProxy

Just after creating a pseudo object, the method makeProxy of class Guarana creates a NewProxy Message and broadcasts it to the meta-configuration of the instantiated class. NewProxy is a subclass of NewObject, whose broadcast allows meta-configurations of classes to reject the creation of proxy Objects by throwing RunTimeExceptions.

If the broadcast terminates successfully, method makeProxy will issue a reconfiguration request to install the meta-object suggested by its caller as the primary meta-object of the created proxy.

3.1.11 InstanceReconfigure

The kernel of **Guaraná** broadcasts a Message of this class to the class of an object, as well as to its superclasses, when it is asked to replace the primary meta-object of an object, but the meta-configuration of the object is `null`.

Class InstanceReconfigure provides methods for obtaining references to the object and to the suggested primary meta-object, as well as for modifying the suggested primary meta-object.

After the Message is broadcast to the meta-configuration of all classes up to root of the class hierarchy, the meta-object in the InstanceReconfigure Message is installed as the primary meta-object, unless it has become `null`.

3.1.12 OperationFactory

An OperationFactory is associated with a single base-level Object, and it can be used by a MetaObject to create Operations addressed to that Object. OperationFactories provide methods for obtaining a reference to the base-level Object and for creating method and constructor invocations, monitor enter and exit, field read and write, array length read, array element read and write, and a special do-nothing operation placeholder (nop).

Operations can be created as replacement or stand-alone ones. In the former case, the Operation to be replaced must be provided as an additional argument, and some validation is performed. In the latter case, an Operation that can be performed independently of any other is created.

3.1.13 OperationFactoryFilter

This class specializes OperationFactory, so that it delegates operation creation requests to another OperationFactory. Its methods can be overridden so as to restrict the set of Operations that can be created.

3.1.14 HashWrapper

Hashtables are a powerful feature of Java, but using them for mapping base-level objects to meta-level data such as operation factories, pending operations, etc, requires some care, because the implementation of Hashtable invokes key object methods such as hashCode and equals.

In order to avoid unintended interactions with base-level objects, a meta-level object should wrap them with HashWrappers, that implement methods hashCode and equals without invoking methods on the base-level object. Note that, if the base-level object specialized any of these methods, the specializations would be disregarded, because HashWrapper simulates the standard implementations of hashCode and equals. A standard implementation of toString is offered too.

3.2 Changes to the Java Virtual Machine

In this section, we describe the modifications we have introduced in *Kaffe OpenVM*, for it to support **Guaraná**.

First of all, every Object had to contain a reference to its MetaObject. For the sake of a simpler implementation, we have decided to add a field to the native description of class Object, instead of trying to encode this reference in modified class descriptors. The main drawback of this approach is that every object is augmented by one word; the main advantage is that checking whether an object is reflective or not (i.e., is associated with a non-null meta-configuration or not) is fast.

Classes are also Objects, thus they may also be associated with MetaObjects. The meta-object associated with a class will receive operations addressed to the class object itself, as well as operations involving `static` methods or fields of the class it represents. The reason for this unification is that `synchronized static` methods acquire locks on the class object. Therefore, if the meta-configuration of a class object could be different from the meta-configuration that handles `static` operations of the corresponding class, the semantics of class monitors in Java would not have been properly modeled in the meta level.

The meta-object reference is hidden from Java programs; it is only accessible in the implementation of the kernel of **Guaraná** and in native code.

The bytecodes that originally just invoked non-`static` methods now check whether the target Object is associated with a MetaObject. If it is, after performing dynamic binding, a method invocation Operation is constructed and performed, that is, operation handling, actual execution and result handling take place. The yielded Result is then un-reified. The bytecode used to invoke `static` methods was also modified, so that the Operation is intercepted only if the class is associated with a MetaObject.

Bytecodes that used to read from and write to fields were changed so that, if the Object (for non-`static` fields) or the class (for `static` fields) is associated with a MetaObject, the Operation is intercepted.

Bytecodes used for indexing arrays, both for reading and for writing, as well as the bytecode for obtaining the length of an array, have been extended so that the array MetaObject can intercept such Operations.

The bytecodes that allocate memory for Objects and arrays were changed so as to support interception of object creation. After they allocate memory for the new Object, they request the primary MetaObject of the creator of the new Object to configure it, a mechanism we have named *meta-configuration propagation*.

The *creator* of an Object is determined based on the method in which the object allocation bytecode appears. If it is a `static` method, the creator is the Class in which the method is declared. Otherwise, the creator is the Object referred to by the keyword `this`, within that method invocation.

After the new Object is configured, a NewObject Message is broadcast to the meta-configuration of the Class of the new Object.

Monitor-related bytecodes, as well as implicit entries and exits of object or class monitors in `synchronized` methods, have been extended so that these Operations can be intercepted.

In addition to bytecodes, the `native` implementation of the Java Core Reflection API and of the Java Native Interface had to be modified to support interception.

4 Performance

We have run some performance tests to try to evaluate the impact of introducing reflective capabilities into a Java interpreter. Our tests have been performed on four different platforms: a single-processor 167 MHz SPARC Ultra 1 (sparc-u1) running Solaris 2.6, a dual-processor 200 MHz SPARC Ultra Enterprise 2 (sparc-u2) running Solaris 2.5, a 100 MHz Pentium (`i586`) running RedHat Linux 5.1, and a 233 MHz Pentium Pro (`i686`) running RedHat Linux 5.0.

On the Solaris platforms, the tests were run in real-time scheduling mode, so as to ensure that no other processes would affect the measured times. On the GNU/Linux platforms, this scheduling mechanism was not available, so we just ensured that the tested hosts were as lightly loaded as possible.

On each host, we have run the same Java program, compiled with Sun JDK's Java compiler, without optimization, to prevent method inlining. The produced bytecodes were executed by different interpreters under different configurations.

Some configurations involve the use of MetaLogger, an example of MetaObject distributed with **Guaraná**, that logs a message to the standard output every time one of its methods is invoked. It is useful for observing the behavior of base-level objects associated with it.

Along the text, we are going to refer to the configurations using the following tags:

`KJ-` Kaffe just-in-time (JIT) compiler without **Guaraná**.

`KJG` Kaffe JIT compiler with **Guaraná**.

`JDK` Sun Java Development Kit (JDK).

`KI-` Kaffe interpreter without **Guaraná**.

`KIG` Kaffe interpreter with **Guaraná**.

`KJN` Kaffe JIT compiler with **Guaraná**, intercepting all operations with a do-nothing meta-object.

`KIN` Kaffe interpreter with **Guaraná**, intercepting all operations with a do-nothing meta-object.

`KJM` Kaffe JIT compiler with **Guaraná**, intercepting and logging all operations, results, messages, initializations and configurations with a MetaLogger.

`KIM` Kaffe interpreter with **Guaraná**, intercepting and logging all operations, results, messages, initializations and configurations with a MetaLogger.

We have used **Guaraná** 1.4.1 and the snapshot of Kaffe 1.0.b1 distributed with it, and Sun JDK 1.1.6 (the official Solaris version and version 4a of the Blackdown Java-Linux port). Kaffe and **Guaraná** were compiled with EGCS 1.1b, with default optimization levels.

For each configuration, we have timed several different operations. Each operation was timed by running it repeatedly inside a loop, with an iteration count large enough for elapsed time in the loop to be greater than 1 second. Each test was run 50 times on each configuration and platform, and the presented values are the average of the runs. All figures are given in seconds. A change in the order of magnitude of the averages obtained is indicated by shifting the figures to the left. As an example, observe Table 1. The program used to perform the tests is the one distributed with **Guaraná** 1.4.1.

We have also measured and averaged the compilation time of the test program itself, in the configurations that do not involve meta-objects, so as to estimate the overall performance impact on a real application caused by introducing the *ability* to intercept operations, without actually intercepting them.

Section 4.19, page 22, contains a summarizing analysis of the data obtained during the tests. In that section, Table 18 and Table 19 are used to present a less fragmented view of the test data obtained so far.

4.1 Empty loop

This test consists exclusively of a loop that decrements a variable until it becomes zero. It does not trigger any reflective mechanism, as it only deals with a variable local to a method. We have included the average times for the empty loop (Table 1) because they can be used as a benchmark for the other tests, as they are also based on measures of the time consumed by a loop that executes the block of instructions pertinent to this test. The times shown in Table 1 correspond to the time for a single iteration in the loop.

Table 1: Empty loop

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	3.1 e−8	2.1 e−8	6.0 e−8	5.0 e−8
KJG	3.1 e−8	2.1 e−8	6.0 e−8	5.0 e−8
JDK	2.1 e−7	2.0 e−7	3.0 e−7	2.4 e−7
KI-	2.0 e−6	6.7 e−7	6.8 e−7	5.6 e−7
KIG	1.2 e−6	5.7 e−7	6.8 e−7	5.6 e−7
KJN	3.1 e−8	2.2 e−8	6.0 e−8	5.0 e−8
KIN	1.2 e−6	5.7 e−7	6.8 e−7	5.6 e−7
KJM	3.1 e−8	2.2 e−8	6.0 e−8	5.0 e−8
KIM	1.2 e−6	5.7 e−7	6.8 e−7	5.6 e−7

4.2 Empty `synchronized` block

The figures presented in Table 2 correspond to the time needed to enter and exit an object's monitor in a loop iteration. In the tests that involve meta-objects, both operations are intercepted.

Table 2: Empty `synchronized` block

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	4.6 e−6	2.1 e−6	4.4 e−6	3.8 e−6
KJG	5.2 e−6	2.4 e−6	5.6 e−6	3.9 e−6
JDK	3.2 e−6	1.3 e−6	1.8 e−6	1.5 e−6
KI-	9.5 e−6	4.3 e−6	7.3 e−6	5.8 e−6
KIG	9.5 e−6	4.3 e−6	7.4 e−6	6.1 e−6
KJN	5.5 e−4	1.8 e−4	3.3 e−4	2.5 e−4
KIN	9.2 e−4	3.0 e−4	4.2 e−4	3.5 e−4
KJM	2.8 e−2	7.5 e−3	1.3 e−2	9.4 e−3
KIM	1.2 e−1	2.5 e−2	3.2 e−2	2.8 e−2

Introducing interception ability in these two operations did not require modification to the definition of bytecodes, as Kaffe implemented `monitorenter` and `monitorexit` by calling C functions through preprocessor macros lockMutex and unlockMutex. These macros were also called just before entering and exiting a `synchronized` method.

We just had to redefine these macros so that alternate functions were called. These functions test whether the object passed as argument is associated with a meta-object or not. If it is, a monitor enter or exit operation object will be created, and method perform of the kernel of **Guaraná** will be invoked to intercept it. If the meta-object reference in the object is `null`, the original lock or unlock function will be called.

4.3 Invoking a `static` method

The numbers presented in Table 3 represent the amount of time spent on an invocation of a `static` method that takes no arguments, and returns `void`. In the last four cases, the class that declares the method is reflective, so the invocation is intercepted. In order to be able to intercept this kind of invocation, we had to modify the definition of bytecode `invokestatic`.

Table 3: Invoking a `static` method

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.4 e−7	9.7 e−8	2.0 e−7	1.5 e−7
KJG	2.6 e−7	1.2 e−7	2.4 e−7	2.0 e−7
JDK	3.1 e−7	1.8 e−7	3.8 e−7	3.1 e−7
KI-	9.8 e−6	4.3 e−6	5.3 e−6	4.8 e−6
KIG	1.1 e−5	4.3 e−6	5.5 e−6	4.4 e−6
KJN	3.0 e−4	9.9 e−5	2.0 e−4	1.5 e−4
KIN	5.9 e−4	1.7 e−4	2.2 e−4	1.8 e−4
KJM	1.2 e−2	3.4 e−3	5.6 e−3	3.5 e−3
KIM	4.0 e−2	1.3 e−2	1.3 e−2	1.0 e−2

In the interpreted case, before invoking recursively the interpreter function to run the `static` method, the new definition of this bytecode will check whether the class whose method is to be invoked is reflective, i.e., if it is associated with a meta-object. If not, normal execution proceeds, otherwise a generic method invocation reification function is invoked. This function takes a pointer to the target object (in this case, since the method is `static`, this pointer is `null`), a pointer to the structure that describes the method to be invoked, a pointer to the top of the stack, onto which the arguments have been pushed, and a pointer to the stack slot where the return value should be stored.

This function tests again whether a meta-object is associated with the target class (or object); in general, a meta-object will be found, and an operation object representing the method invocation will be created and performed. In order to create the operation object, the argument list must be copied, so we must first parse the signature of the method in order to find out how many stack slots the argument list takes, so that we do not copy too much or too little. After the execution of the operation, the result is stored in the provided stack slot.

In the case of the just-in-time compiler, native code is generated so that, before pushing the method arguments onto the stack, the target class is tested for the existence of a meta-object. If the method does not need to be intercepted, normal method invocation takes place, otherwise, arguments are pushed onto the stack with an offset of two words, so that a reference to the method to be called and a reference to the target object (`null`, in this case) can be passed as the first two arguments. Then, an interception function is selected, based on the return type of the called method.

These interception functions all call a generic method invocation function provided by Kaffe. This function tests if the target class (or object) is reflective, and generates a direct invocation of the target method or an invocation of yet another interception function, but now a generic one. This function takes a pointer to the target object's meta-object, a pointer to the target object, a pointer to the method to be invoked, a `va_list`[3] containing the arguments to be passed to the method, and a pointer to a `union` where the result of the method should be stored. This function assumes the meta-object pointer is non-`null`, so it always reifies the invocation. First, it parses the method signature to find out how many stack slots it takes, then it allocates a memory area of appropriate size, then it copies the method arguments into this area, parsing the signature again to use the appropriate types to read the argument values from the `va_list`.

After the operation is performed, the result is stored in the provided `union`. The generic method invocation function returns this value as a `union`, and the type-specific interception function extracts the correct return value from this `union` and returns it.

4.4 Invoking a `private` method

Table 4 shows the cost of invoking a non-`static` `private` method of a potentially reflective object. The `invokespecial` bytecode, used in this kind of invocation, is also used for invoking constructors and, in some cases, `final` methods. It is the fastest non-`static` invocation because it is statically bound. The invoked method, in our test case, has no arguments besides the implicit `this`, and returns `void`.

The implementation of this bytecode was modified almost exactly as the previous one, as the address of the method to be called is also known at JIT-compilation time. The only difference is that, instead of passing `null` as the place-holder for the pointer to the target object, the actual target object is passed.

4.5 Invoking a non-`final` method

Non-`private` non-`static` methods declared in classes (i.e., not in `interfaces`) are dynamically bound on a per-object basis. Kaffe implements dynamic binding using a per-class dispatch table, so that the element of the dispatch table corresponding to an overridden method points to the most derived overrider. In Table 5, we present the amount of time

[3] A `va_list` is a standard C structure that allows a function to accept a variable number of arguments.

Table 4: Invoking a `private` method

Conf	`i586`	`i686`	`sparc-u1`	`sparc-u2`
`KJ-`	1.6 e−7	1.1 e−7	2.0 e−7	1.5 e−7
`KJG`	3.5 e−7	1.2 e−7	2.4 e−7	2.0 e−7
`JDK`	3.6 e−7	1.9 e−7	4.7 e−7	3.8 e−7
`KI-`	9.2 e−6	4.3 e−6	5.5 e−6	4.9 e−6
`KIG`	1.2 e−5	4.6 e−6	7.6 e−6	4.4 e−6
`KJN`	3.2 e−4	1.0 e−4	1.8 e−4	1.4 e−4
`KIN`	6.5 e−4	1.7 e−4	2.3 e−4	1.9 e−4
`KJM`	1.5 e−2	4.5 e−3	7.4 e−3	5.5 e−3
`KIM`	5.9 e−2	1.8 e−2	1.8 e−2	1.9 e−2

spent on invoking, with the `invokevirtual` bytecode, a do-nothing method that takes only the implicit `this` argument and returns void.

Table 5: Invoking a non-`final` method

Conf	`i586`	`i686`	`sparc-u1`	`sparc-u2`
`KJ-`	1.2 e−6	4.8 e−7	6.2 e−7	5.2 e−7
`KJG`	1.6 e−6	1.3 e−6	5.8 e−7	5.2 e−7
`JDK`	7.9 e−7	5.5 e−7	7.9 e−7	6.3 e−7
`KI-`	1.1 e−5	4.6 e−6	5.7 e−6	5.3 e−6
`KIG`	1.3 e−5	4.6 e−6	6.2 e−6	4.8 e−6
`KJN`	3.3 e−4	1.1 e−4	2.0 e−4	1.5 e−4
`KIN`	6.5 e−4	1.7 e−4	2.4 e−4	1.9 e−4
`KJM`	1.4 e−2	4.7 e−3	7.9 e−3	5.6 e−3
`KIM`	5.9 e−2	1.8 e−2	2.0 e−2	1.7 e−2

The greatest difficulty for intercepting this bytecode was that, in JIT compiler mode, the dispatch table only contained pointers to the native code generated for each method, but **Guaraná** needed pointers to the structures that describe methods. So, we modified the format of the dispatch table, so as to accommodate our needs: it has become twice as large, in JIT mode, because it contains pointers both to the native code and to the method structure. If the target object is not reflective, the pointer to native code is loaded from the dispatch table, otherwise, the pointer to the method structure is used to intercept the method invocation, just like in the other invocation bytecodes.

4.6 Invoking an `interface` method

When the static type of an object (i.e., the type known at compile-time) is an `interface` one, the bytecode used for a method invocation is `invokeinterface`. Dynamic binding is much more expensive than in the `invokevirtual` case, because interface methods cannot share a common index in a dispatch table. Hence, for every invocation, the requested method name and signature must be looked up in the object's class, as well as in its superclasses. Although, in our example, dynamic binding ends up selecting the same method of the object used in the previous test, the dynamic binding takes much longer, as we can observe in Table 6.

Table 6: Invoking an `interface` method

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	2.9 e−6	1.2 e−6	1.8 e−6	1.5 e−6
KJG	3.1 e−6	1.2 e−6	1.8 e−6	1.6 e−6
JDK	1.2 e−6	8.5 e−7	9.2 e−7	8.0 e−7
KI-	1.4 e−5	6.0 e−6	7.8 e−6	7.2 e−6
KIG	1.3 e−5	5.6 e−6	9.4 e−6	6.4 e−6
KJN	3.3 e−4	1.1 e−4	1.9 e−4	1.5 e−4
KIN	6.7 e−4	1.8 e−4	2.4 e−4	1.9 e−4
KJM	1.9 e−2	4.7 e−3	7.4 e−3	5.6 e−3
KIM	5.9 e−2	1.8 e−2	2.0 e−2	1.5 e−2

Once again, the data provided by the JIT compiler runtime was not enough for intercepting method invocations correctly: the function that would look up the `interface` method and signature in the object's class and its superclasses would return a pointer to the native code of the selected method, but we needed a method structure. Thus, we have modified the look up function, so that it would return the method structure and, if the method did not have to be intercepted, we would load the address of the native code from the method structure, just as the look up function would have done.

4.7 Loading a `static` field

In Table 7, we show how long it takes to load the value of a `static int` variable from of a potentially reflective class into a local variable.

The Kaffe interpreter implements the `getstatic` bytecode by looking up the address of the field in the field description structure and pushing its value onto the stack. A `switch` statement selects the appropriate number of stack slots to allocate and the size of the data to be copied. Moving the loaded value from the stack to the local variable takes another `bytecode`, that is not modified at all.

For the `getstatic` bytecode to support interception, we have inserted the meta-object test just before the `switch` statement; if no meta-object is available, the original code is

Table 7: Loading a `static` field

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.4 e−7	3.1 e−8	6.6 e−8	5.5 e−8
KJG	2.3 e−7	7.7 e−8	1.7 e−7	1.5 e−7
JDK	3.2 e−7	2.8 e−7	4.7 e−7	3.7 e−7
KI-	4.7 e−6	1.8 e−6	2.2 e−6	1.8 e−6
KIG	4.6 e−6	1.8 e−6	2.6 e−6	1.8 e−6
KJN	2.7 e−4	9.1 e−5	1.7 e−4	1.3 e−4
KIN	6.1 e−4	1.6 e−4	2.2 e−4	1.7 e−4
KJM	1.1 e−2	2.9 e−3	4.5 e−3	3.3 e−3
KIM	4.4 e−2	9.7 e−3	1.1 e−2	1.2 e−2

executed, otherwise, we run another `switch` statement that just allocates stack space for the field to be loaded, and call a generic field load interception function, passing to it a pointer to the class' meta-object, a pointer to the class structure, a pointer to the class where the field is declared (that is the same as the previous one, but is needed for non-`static` fields), a pointer to the field structure, and a pointer to the stack slot where the value of the field should be stored. This function assumes the meta-object is non-`null`, so it just creates an operation object and performs it, storing the result of the operation into the provided stack slot.

The JIT compiler is much faster, as it encodes the address of the field in the generated code, and it selects the appropriate load and store instructions at compile-time. Furthermore, the value loaded into a register needs not be immediately stored in the stack frame, if it is going to be used in some other computation. Due to limitations in the JIT compiler, though, the register that represents the Java stack slot cannot be the same that represents the local variable: they represent different native stack slots. Furthermore, sooner or later, the registers have to be spilled onto their native stack slots, even if they are not going to be used in the future: Kaffe does not currently perform any kind of global analysis.

Introducing interception abilities in this bytecode has a rather high cost, because the test for meta-object introduces new basic blocks in the program. Since Kaffe does not perform global register allocation, it resets the state of all registers at the beginning of every basic block and spills them all at block's end. A simple optimization has allowed us not to reset the register states before the field load operation, despite the branch just before it, but, nevertheless, all registers are spilled after the field load, and, when the two branches merge back, all registers are reset, so that the field value must be loaded back from the stack if it is going to be used in the next few instructions.

In order to intercept field load operations, different interception functions are used for different field types. They take all arguments the generic field load interception function take, except the pointer to the result stack slot: the result is returned by the type-specific functions. They just call the generic field load interception function, passing them a pointer to a local stack slot, then return the appropriate value extracted from this slot.

4.8 Writing to a `static` field

Table 8 gives the time needed to write the value of a zero-valued local variable in a `static int` field of a potentially reflective class.

Table 8: Writing to a `static` field

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.3 e−7	3.4 e−8	6.6 e−8	5.5 e−8
KJG	3.5 e−7	6.6 e−8	1.3 e−7	1.1 e−7
JDK	3.4 e−7	3.0 e−7	4.8 e−7	3.8 e−7
KI-	5.4 e−6	1.9 e−6	2.2 e−6	1.8 e−6
KIG	4.1 e−6	1.8 e−6	2.7 e−6	1.9 e−6
KJN	2.8 e−4	9.1 e−5	1.7 e−4	1.3 e−4
KIN	5.9 e−4	1.6 e−4	2.1 e−4	1.7 e−4
KJM	1.1 e−2	3.2 e−3	4.7 e−3	3.6 e−3
KIM	3.8 e−2	1.0 e−2	1.5 e−2	1.2 e−2

The modified bytecode, in this case, is `putstatic`. In the Kaffe interpreter implementation, it would just perform a `switch` statement, write the value on top of the stack onto the field address, and pop it from the stack. Our modified implementation includes a meta-object existence test. If a non-`null` meta-object is associated with the class that declares the `static` field, a generic field write interception function is called. This function takes the same arguments that the generic field load interception function expects, but, in this case, the pointer to the stack slot contains the value to be written.

In the JIT compiler case, all the optimizations and overheads presented in the previous section apply.

4.9 Loading a non-`static` field

The times needed to obtain the value of a field of type `int` from an object, that may be reflective, and store it in a local variable, are displayed in Table 9.

One of the two differences between the `getfield` bytecode, used in this case, and `getstatic`, already described, has to do with the arguments passed to the interception functions: in this case, the second argument is the object whose field is going to be loaded, instead of a duplicated pointer to the class object.

The other difference is that, in the `static` case, the addresses of the class and of the field are known at compile-time, so they are treated as constants in the compiled code; in the non-`static` case, the address of the class is still used, as the third argument to the interception function, but the address of the object is only known at execution time, and, instead of the absolute address of the field, the field offset is encoded in the compiled code.

Table 9: Loading a non-`static` field

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	6.9 e−8	4.1 e−8	6.0 e−8	5.0 e−8
KJG	2.7 e−7	7.6 e−8	1.5 e−7	1.3 e−7
JDK	3.5 e−7	2.5 e−7	5.4 e−7	4.4 e−7
KI-	7.2 e−6	2.0 e−6	2.4 e−6	2.0 e−6
KIG	5.6 e−6	2.0 e−6	2.9 e−6	2.0 e−6
KJN	2.8 e−4	9.3 e−5	1.7 e−4	1.3 e−4
KIN	6.1 e−4	1.6 e−4	2.1 e−4	1.7 e−4
KJM	1.2 e−2	4.7 e−3	6.9 e−3	5.4 e−3
KIM	5.5 e−2	1.5 e−2	1.7 e−2	1.6 e−2

4.10 Writing to a non-`static` field

Table 10 lists the duration of an operation that stores the value of a local variable, initialized to zero, into an `integer` field of an object whose meta-object may be non-`null`.

Table 10: Writing to a non-`static` field

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.3 e−7	3.1 e−8	5.4 e−8	4.5 e−8
KJG	2.5 e−7	6.1 e−8	9.0 e−8	7.5 e−8
JDK	3.6 e−7	2.9 e−7	5.5 e−7	4.4 e−7
KI-	6.7 e−6	2.1 e−6	2.4 e−6	2.0 e−6
KIG	4.9 e−6	2.0 e−6	3.0 e−6	2.1 e−6
KJN	2.9 e−4	9.2 e−5	1.7 e−4	1.3 e−4
KIN	5.9 e−4	1.6 e−4	2.1 e−4	1.7 e−4
KJM	1.6 e−2	3.9 e−3	6.4 e−3	5.2 e−3
KIM	5.7 e−2	1.5 e−2	1.8 e−2	1.6 e−2

One important difference between the `putfield` bytecode and the other field-related bytecodes is that the stack position of the object whose field is to be written to depends on the type of the field. For this reason, what used to be a single `switch` statement in the implementation of Kaffe has been split into two separate ones: first, we find out where in the stack the target object of the operation is located, so we can check whether it is reflective or not. If it is not, the original `switch` statement stores the top of the stack onto the object's field. Otherwise, a field write interception function is called, just like the `putstatic` does, except that the second argument is a pointer to the object, not to the class. The additional `switch` statement affects only the interpreter, because, in the JIT compiler, it is only evaluated at compile-time. Nevertheless, the discussions in the previous sections apply to this bytecode too.

4.11 Loading the `length` of an array

The time needed to load the `length` of a possibly reflective array of `int`, of `length` 1, into a local variable, is presented in Table 11.

Table 11: Loading the `length` of an array

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	6.9 e−8	4.1 e−8	6.0 e−8	5.0 e−8
KJG	2.5 e−7	7.7 e−8	1.4 e−7	1.3 e−7
JDK	3.7 e−7	2.7 e−7	5.4 e−7	4.3 e−7
KI-	2.9 e−6	1.2 e−6	1.3 e−6	1.1 e−6
KIG	2.4 e−6	1.1 e−6	1.3 e−6	1.2 e−6
KJN	2.7 e−4	8.8 e−5	1.7 e−4	1.3 e−4
KIN	5.6 e−4	1.5 e−4	2.1 e−4	1.7 e−4
KJM	8.4 e−3	3.1 e−3	4.5 e−3	3.4 e−3
KIM	4.3 e−2	9.7 e−3	1.4 e−2	9.9 e−3

Although the `length` of an array is not properly a field of the array object, the code produced by the original JIT compiler for the `arraylength` and the `getfield` bytecodes is identical, as the length of an array is stored in a fixed offset of the object that represents the array. On the interpreter, however, `arraylength` operation is much faster `getfield`, because the offset of the `length` is known at interpreter compile-time, it does not have to be looked up in a field structure.

As usual, we have added the meta-object existence test before the execution of the regular array length operation. If the array is found to be reflective, both engines just call a function that takes a pointer to the array object and returns the array length. This function tests again whether the array is reflective. If its meta-object has become `null`, the length of the array is returned immediately. Otherwise, an operation object is created and performed, then its result is returned. The JIT performance incurs in the register spilling and reloading overhead in this case too.

4.12 Loading an element of an array

Storing in a local variable the first element of the array used in the previous section takes the amount of time displayed in Table 12.

There are different array element load operations for each primitive type, and yet another for object types. However, their implementations are identical, except for the calculation of the offset from the beginning of the array and the actual element load instruction. Therefore, there is no need to time all possible array operations. We have probably selected an `int` array, accessed through the `iaload` bytecode, so as to save typing, since `int` is the shortest type name in Java. The fact that it fits exactly in the registers of the tested platforms has just made the results look worse, because there is no need for any conversion that might have reduced the relative overhead introduced by **Guaraná**.

Table 12: Loading an element of an array

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.1 e−7	5.2 e−8	1.1 e−7	7.7 e−8
KJG	3.3 e−7	1.0 e−7	1.7 e−7	1.5 e−7
JDK	3.7 e−7	2.2 e−7	6.4 e−7	5.1 e−7
KI-	8.3 e−6	1.4 e−6	1.5 e−6	1.3 e−6
KIG	2.9 e−6	1.3 e−6	1.5 e−6	1.3 e−6
KJN	2.8 e−4	9.0 e−5	1.7 e−4	1.3 e−4
KIN	5.6 e−4	1.5 e−4	2.1 e−4	1.7 e−4
KJM	7.8 e−3	2.2 e−3	3.8 e−3	3.2 e−3
KIM	3.8 e−2	9.4 e−3	1.0 e−2	9.8 e−3

We have introduced a meta-object existence test in all array load bytecodes *before* the array bound check, so that the meta-level can make arrays seem larger than they actually are. If a meta-object is associated with the array, interception occurs. In the interpreter case, a generic array load interception function is called. It takes, as arguments, a pointer to the array meta-object, a pointer to the array itself, the index of the array element to be loaded, and the stack slot where it should be stored. The JIT, on the other hand, uses specialized functions for each different type, that return the loaded values. Because of the additional basic blocks, register spilling and reloading becomes necessary.

4.13 Writing to an element of an array

The figures in Table 13 represent the time spent by an operation that stores the value of a zero-initialized local variable into the first element of the aforementioned array.

Table 13: Writing to an element of an array

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	9.9 e−8	5.6 e−8	8.9 e−8	7.2 e−8
KJG	3.3 e−7	8.7 e−8	1.3 e−7	1.1 e−7
JDK	4.2 e−7	3.1 e−7	6.4 e−7	5.1 e−7
KI-	3.4 e−6	1.4 e−6	1.5 e−6	1.2 e−6
KIG	2.9 e−6	1.3 e−6	1.5 e−6	1.3 e−6
KJN	2.9 e−4	9.1 e−5	1.7 e−4	1.3 e−4
KIN	5.5 e−4	1.6 e−4	2.1 e−4	1.7 e−4
KJM	9.3 e−3	2.5 e−3	4.5 e−3	3.3 e−3
KIM	3.7 e−2	9.9 e−3	1.4 e−2	9.4 e−3

The only difference between the array load and the array store bytecodes is the direction of the array element data. In this case, the interpreter calls a generic array store interception function that takes a pointer to the stack slot that contains the value to be stored, and the JIT compiler calls type-specific interception functions that take the value to be stored as an additional argument. Register management overhead due to additional basic blocks applies too.

4.14 Creating objects

Creating an object involves allocating memory for an object and invoking its constructor, that are two separate operations at bytecode level. Furthermore, if we were to run a test program to time this operation, it would inevitably be influenced by the cost of garbage collection. Since the garbage collector cannot be controlled reliably, because of different object sizes and increased size of JIT-generated code, we have decided not to run this test.

The first bytecode involved in the creation of an object is `new`, that allocates the amount of memory necessary for an object of a specified class and initializes all its fields with zeroes and `null`s, except its dispatch table, that is initialized to point to the dispatch table of its class. This bytecode is implemented as an invocation of a function that takes a pointer to a class and returns a pointer to the newly created object. We have modified this bytecode so that it calls an alternate function that takes an additional argument: a pointer to the creator of the new object. After this function calls the original object creation function, it calls a generic meta-configuration propagation one.

This propagation function checks whether the creator has a meta-object. If it does, it invokes method configure of that meta-object, then sets the new object's meta-configuration to the meta-object returned by this method. Afterwards, if the object's class has a non-`null` meta-configuration, a NewObject message is created and broadcast to the class' meta-configuration, by invoking method broadcast of the kernel of **Guaraná**.

If neither the creator nor the class are reflective, the reflection overhead in bytecode `new` is minimal, and it is the same for both the interpreter and the JIT compiler.

The second bytecode involved in object creation is `invokespecial`, used to invoke the new object's constructor. We have already presented the overhead introduced in this operation in Table 4.

4.15 Creating arrays

There are two different bytecodes for creating arrays: one that creates arrays of primitive types, and another that creates arrays of class types. The implementation of these bytecodes is very similar, and both were modified exactly like bytecode `new`: the array creation function calls gained an additional argument, a pointer to the array creator, and were changed so as to call alternate functions that supported this additional argument. After invoking the original array creation functions, they would call the generic meta-configuration propagation function. Since arrays do not have constructors, the additional overhead due to constructor invocation does not exist.

4.16 Creating arrays of arrays

The `multianewarray` bytecode creates multi-dimensional arrays as a single operation. This bytecode was modified in a slightly different manner: instead of creating an alternate function that would call the original multi-array creation one, we have modified the function itself, so that it would support an additional argument, a pointer to the array creator.

The rationale for this difference is that, just after creating the top-level array, the creator's meta-configuration must be propagated into this array, so that, when the sub-arrays are created, the meta-configuration of the container array is already set up to propagate into them.

4.17 Printing a `String`

As a first attempt to measure the overall impact of the introduction of reflection, we have measured how long it takes for the System.err object to print the `String` `‘‘Hello world!’’` and skip to the next line. The obtained times are listed in Table 14.

Table 14: Printing a `String`

Conf	`i586`	`i686`	`sparc-u1`	`sparc-u2`
`KJ-`	2.8 e−4	8.2 e−5	1.4 e−4	1.1 e−4
`KJG`	4.1 e−4	8.7 e−5	1.5 e−4	1.2 e−4
`JDK`	3.7 e−4	1.9 e−4	2.2 e−4	1.8 e−4
`KI-`	1.6 e−3	4.3 e−4	5.8 e−4	5.1 e−4
`KIG`	1.8 e−3	4.5 e−4	6.1 e−4	5.0 e−4
`KJN`	3.7 e−4	9.7 e−5	1.6 e−4	1.3 e−4
`KIN`	1.8 e−3	4.6 e−4	5.8 e−4	4.8 e−4
`KJM`	3.5 e−4	1.0 e−4	1.4 e−4	1.1 e−4
`KIM`	1.8 e−3	4.7 e−4	5.8 e−4	5.2 e−4

No objects are created in this operation, so no garbage collection takes place. Furthermore, since neither class System nor object System.err are reflective, no interception takes place.

4.18 Compiling a program

Timing the compilation of the test program with the various available interpreters has produced the figures in Table 15. We have not timed the executions with meta-objects, because, at this point, we are only interested in measuring the overall performance penalty introduced by the potential of intercepting operations.

On short-running applications like this, most of the time is spent on virtual machine initialization and JIT compilation, not on running the application itself. The virtual machine start-up, for example, involves executing very large array initialization methods, whose JIT-

Table 15: Compiling a program: total execution time

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.7 e+1	5.1 e+0	9.1 e+0	7.5 e+0
KJG	2.3 e+1	7.2 e+0	1.2 e+1	9.6 e+0
JDK	4.8 e+0	2.1 e+0	2.2 e+0	1.9 e+0
KI-	3.0 e+1	9.2 e+0	1.3 e+1	1.1 e+1
KIG	3.2 e+1	9.4 e+0	1.3 e+1	1.0 e+1

compilation wastes a lot of memory and CPU cycles, because these methods are executed only once.

Although a complex program, involving several similar classes, is being compiled, Table 16 shows that more than 50% of the total time was spent on JIT-compiling Java Core classes and the Java compiler itself. Table 17 presents the differences between the total time and the JIT-compilation time, that represents the time spent on running the actual application. Hence, long running applications, that repeatedly run the same methods, should present a reflection overhead similar to the relative overhead of this table.

Table 16: Compiling a program: JIT compilation time

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	3.9 e+0	1.3 e+0	1.8 e+0	1.9 e+0
KJG	8.0 e+0	2.8 e+0	3.3 e+0	2.9 e+0

Table 17: Compiling a program: disregarding JIT-compilation time

Conf	i586	i686	sparc-u1	sparc-u2
KJ-	1.3 e+1	3.8 e+0	7.3 e+0	5.5 e+0
KJG	1.6 e+1	4.5 e+0	8.8 e+0	6.7 e+0

4.19 Overall analysis

In Table 18, we present the relative slow down caused by adding interception code to the Kaffe interpreter on all tested platforms. Table 19 contains the corresponding data for the Kaffe JIT compiler. The listed figures are calculated from numbers with a higher precision than the ones presented in the previous tables, so that rounding of those values does not affect the figures in this table.

Table 18: Overall analysis (interpreter engine)

Table Number and Mnemonic	i586	i686	sparc-u1	sparc-u2
1: emptyloop	−41%	−15%	−0%	−0%
2: monitorenter/exit	−0%	+1%	+0%	+4%
3: invokestatic	+13%	+0%	+4%	−8%
4: invokespecial	+30%	+8%	+38%	−10%
5: invokevirtual	+17%	−0%	+7%	−9%
6: invokeinterface	−3%	−7%	+20%	−10%
7: getstatic	−3%	−2%	+20%	−0%
8: putstatic	−23%	−3%	+24%	+4%
9: getfield	−22%	−2%	+19%	−0%
10: putfield	−26%	−2%	+25%	+6%
11: arraylength	−18%	−9%	+2%	+12%
12: iaload	−64%	−6%	+1%	−0%
13: iastore	−14%	−3%	+1%	+1%
14: println	+6%	+4%	+3%	−2%
15: compile	+5%	+2%	−2%	−3%

The table only compares executions that do not involve meta-objects, because, when a meta-object intercepts an operation, the cost of the operation grows by some orders of magnitude. In fact, intercepting a simple operation involves dozens of method invocations, some of them implemented in native code that calls Java code. In addition to the fact that Kaffe interface for calling Java from native code is very slow, we should also consider that every intercepted operation causes the creation of an operation object and a result object, that must be garbage collected, and garbage collection is slow and unpredictable.

In certain combinations of platform and engine, an operation executes faster on **Guaraná** than on the corresponding combination without it. This is quite hard to explain, since **Guaraná** always executes at least as much code as Kaffe does. The tests have been verified so as to ensure that the results are correct, and the generation of the tables from the test runs is totally automated, so there is no place for human error. The better performance can be attributed to factors such as improved fast-RAM cache hit ratio or alignment issues.

The overhead introduced by interception on the interpreter engine is mostly small, because the interpreter is usually orders of magnitude slower than the test for existence of a meta-object. The JIT, however, is severely affected by increased register pressure and additional register spilling and reloading. JIT-compilation costs have increased too, as our tests have shown, but they have only affected the figures of the `compile` test. In all other cases, we ensure that a method is JIT-compiled before we start timing its execution.

Although the interception code has introduced moderate penalties for invoking `static` and `private` methods, the most common kind of invocation (non-`final`) causes a very small overhead, except on `i686`, and `interface` invocations are almost not affected at all.

The bad results for some invocation bytecodes on one `x86` platform but not on the other is unexpected, considering that it executes *exactly* the same code on both. It looks like these tests introduce pathological pipeline stalls or branch prediction errors that degrade

Table 19: Overall analysis (JIT-compiler engine)

Table Number and Mnemonic	i586	i686	sparc-u1	sparc-u2
1: emptyloop	+0%	+1%	+0%	+0%
2: monitorenter/exit	+12%	+10%	+27%	+3%
3: invokestatic	+91%	+20%	+23%	+34%
4: invokespecial	+119%	+8%	+19%	+28%
5: invokevirtual	+30%	+158%	−6%	+0%
6: invokeinterface	+7%	+2%	+3%	+2%
7: getstatic	+68%	+148%	+163%	+163%
8: putstatic	+180%	+97%	+90%	+90%
9: getfield	+293%	+86%	+149%	+149%
10: putfield	+103%	+96%	+66%	+66%
11: arraylength	+258%	+86%	+140%	+150%
12: iaload	+191%	+98%	+55%	+95%
13: iastore	+236%	+55%	+41%	+45%
14: println	+45%	+6%	+5%	+12%
15: compile	+36%	+42%	+32%	+29%
16: compile-JIT	+105%	+112%	+81%	+54%
17: compile-diff	+16%	+17%	+20%	+20%

performance, since the average penalty, measured in `compile-diff`, is very similar on both `x86` platforms, and much lower than most of the individual penalties.

On the other hand, the bad results for all load and store operations on the JIT engines are expected, for the reasons already presented. Fortunately, in object-oriented applications, field and array operations are usually intertwined with method invocations and object creations. Since the latter operations incur a much smaller penalty, and they are one order of magnitude slower than the former ones, the net performance penalty may be acceptable, as the introduction of reflective capabilities may pay off.

5 Future optimizations

The reflection overhead on the interpreter is almost negligible, so there is very little need to worry about optimizing it any further. For the JIT code, there is little hope for similarly small overheads, though. One possible approach would be to implement all operations, even field and array ones, as invocations of dynamically generated JIT-compiled code. Then, instead of having to load the meta-object reference before performing an operation, an extended dispatch table would contain pointers to these JIT-generated functions, on non-reflective objects, and to interceptor functions, in the case of reflective objects.

Unfortunately, we do not think this solution would do very well: first, because we would have to look up the dispatch table before executing every single operation, and the virtual method invocation cost is currently much higher than non-virtual method invocation, so we would end up increasing the cost of most operations, instead of reducing it.

Furthermore, invoking a function requires saving most registers on some ABIs, but this is not required when contents of memory addresses are loaded directly, as field load operations are currently implemented. In fact, the way **Guaraná** is currently implemented means that, whenever a field or array operation is performed, registers must be saved because it *might* be necessary to invoke an interceptor function. A promising optimization involves not saving registers at all in case no interception is necessary, and modifying the interception code so that it leaves registers just like the original code would. This would decrease the cost of both branches, because they currently save all registers and mark them all as unused before they join to proceed to the next instruction. Furthermore, if the JIT compiler ever gets smarter about global register allocation, the additional branches introduced by **Guaraná** will not get it confused.

One of the reasons why actually intercepting an operation is slow is that it always involves creating two objects: the reified operation and the reified result. We might think of optimizing away the instantiation of these objects, by defining specialized interceptor (handle) methods for different kinds of operations and results. However, this would complicate the meta-object protocol without solving the problem, because then, instead of instantiating operations and results, it would be necessary to dynamically create instances of Method and Field.

However, it might be possible to optimize away the construction of a result object, if it is not requested. In the case of method invocations, we could also try to allocate a single chunk of memory, to contain both the operation object and the argument list, instead of allocating two chunks.

The most important optimization is in the native method invocation interface. Whenever a method is invoked from native code, its signature is parsed several times, so as to calculate its size, build an argument list, fill it in, then actually invoke the method. A lot of this work could be done in advance, and, in the JIT engine, it would be possible to generate specialized dispatchers, fragments of code that would take an argument list in a standard format and call the desired method with the arguments properly converted to the calling convention. In case of intercepting a method invocation, we could also have a pre-compiled interceptor, that would reify the invocation much faster than the current code.

6 Conclusions

The implementation of the reflective architecture of **Guaraná** required some modifications in a Java interpreter, but not in the Java programming language. Thus, any program created and compiled with any Java compiler will run on our implementation, and it will be possible to use reflective mechanisms in order to extend them.

Unfortunately, our implementation depends on a particular interpreter, but we can prove it is *impossible* to implement our meta-object protocol transparently in 100% Pure Java, due to limitations related with `native` methods and bytecode verification.

Our modifications have reduced the speed of the interpreter, but we believe the flexibility introduced by the reflective capabilities outweighs this inconvenience. Furthermore, the performance impact analysis has revealed the current hot spots in the interception mechanisms. We expect to reduce this impact by implementing the suggested optimizations.

A Obtaining Guaraná

Additional information about **Guaraná** can be obtained in the Home Page of **Guaraná**, at the URL `http://www.dcc.unicamp.br/~oliva/guarana/`. The source code of its implementation atop of the *Kaffe OpenVM*, on-line documentation and full papers are available for download. **Guaraná** is *Free Software*, released under the GNU General Public License, but its specifications are open, so non-free clean-room implementations are possible.

B Acknowledgments

This work is partially supported by FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*), grants 95/2091-3 for Alexandre Oliva and 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*).

Special thanks to Tim Wilkinson, for having started the development of Kaffe and having released it as free software.

References

[1] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Java Series. Addison–Wesley, September 1996. Version 1.0.

[2] Pattie Maes. Concepts and experiments in computation reflection. *ACM SIGPLAN Notices*, 22(12):147–155, December 1987.

[3] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The reflexive architecture of Guaraná. Technical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, April 1998.

[4] Brian C. Smith. Prologue to "Reflection and Semantics in a Procedural Language". PhD Thesis Prologue, 1985.