The Design and Implementation of **Guaraná**

Alexandre Oliva
oliva@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

Abstract

Several reflective architectures have attempted to improve meta-object reuse by supporting composition of meta-objects, but have done so using limited mechanisms such as Chains of Responsibility. We advocate the adoption of the Composite pattern to define *meta-configurations*. In the meta-object protocol (MOP) of **Guaraná**, a *composer* meta-object can control *reconfiguration* of its component meta-objects and their interactions with base-level objects, resolving conflicts that may arise and establishing meta-level *security policies*.

Guaraná is currently implemented as an extension of *Kaffe OpenVM*$^{\text{TM}}$, a free implementation of the Java[1] Virtual Machine. Nevertheless, most design decisions presented in this paper can be transported to other programming languages and MOPs, improving their flexibility, reconfigurability, security and meta-level code reuse. We present performance figures that show that it is possible to introduce run-time reflection support in a language like Java without much impact on execution speed.

1 Introduction

Object-oriented design is based on abstraction and information hiding (encapsulation). These concepts have provided an effective framework for the management of complexity of applications. Within this framework, software developers strive to obtain applications that are highly coherent and loosely coupled. Unfortunately, object orientation alone does not address the development of software that can be easily adapted.

The concept of open architectures [6, 7] has been proposed as a partial solution to the problem of creating software that is not only modular, well-structured, but also easier to adapt. Open architectures encourage a modular design where there is a clear separation of *policy*, that is, *what* a module has been designed for, from the *mechanisms* that implement a policy, that is, *how* a policy is materialized. The implementation of system-oriented mechanisms such as concurrency control, distribution, persistence and fault-tolerance can benefit from this approach to software construction.

Computational reflection [13, 21] (henceforth just reflection) has been proposed as a solution to the problem of creating applications that are able to maintain, use and change representations of their own designs (structural or behavioral). Reflective systems are able to use self-representations to extend and *adapt* their computation. Due to this property, they are being used to implement open software architectures. In reflective architectures, components that deal with the processing of self-representation and management of an application reside in a software layer called *meta-level*. Components that deal with the functionality of the application are assigned to a software layer called *base-level*. In object-oriented reflective systems, meta-level objects that implement management policies are called *meta-objects*.

Due to their inherent structure, the existing reflective architectures and MOPs may induce developers to create complex meta-objects that, in an all-in-one approach, implement many management aspects of an application or, alternatively, to construct coherent but tightly coupled meta-objects. Both alterna-

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

tives make reuse, maintenance and adaptation of an application harder, especially of its meta-level, the layer in which most of the adaptations tend to occur in an open architecture.

In contrast, **Guaraná** [20] allows meta-objects to be combined through the use of *composer*s. Composers [17] are meta-objects that can be used to define arbitrary policies for delegating control to other meta-objects, including other composers. They provide the glue code to combine meta-objects, and to resolve conflicts between incompatible ones. The use of composers encourages the separation of the *structure* of the meta level from the *implementation* of individual management aspects.

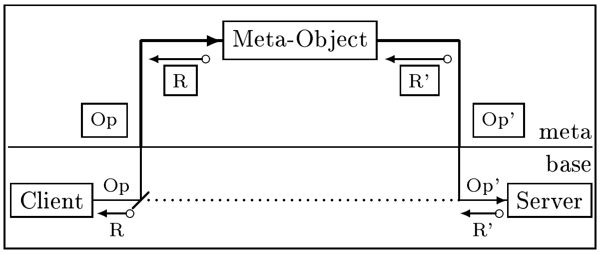
Our implementation of **Guaraná**, based on a Java interpreter that supports just-in-time compilation, has shown that it is possible to introduce interception mechanisms, essential for the deployment of behavioral reflection, with a small overhead. We believe that this overhead is a minor drawback, when compared with the flexibility introduced by our MOP.

This paper is structured as follows. In the next section, we discuss some related works. In Section 3, we present the reflective architecture of **Guaraná**. Section 4 contains a short description of our implementation of this architecture, extending a freely-available Java Virtual Machine. In Section 5, we present some figures about the impact of **Guaraná** on the performance of applications. Section 6 lists some possible future optimizations for our implementation of **Guaraná**. Finally, in Section 7, we summarize the main points of the paper.

2 Related Work

The development of generic mechanisms for the composition of meta-objects is still in its initial stages. OpenC++ [2] does not provide direct support for composition. MOOSTRAP [16] and MetaXa [9] (formerly known as MetaJava) support sequential composition of *similar* meta-objects. We say that meta-objects are similar if they implement the same interface.

Apertos [23] and CodA [14] assign aspects of base-level execution, such as sending, receiving and



When a Client requests an operation Op from a Server object, the operation is intercepted, reified (represented as an object) and presented to a Meta-Object. It may choose to deliver a different operation Op' to the Server, obtaining the result R', that is also reified. Having delivered an operation or not, it must reply with a result R, that is unreified and returned to the Client.

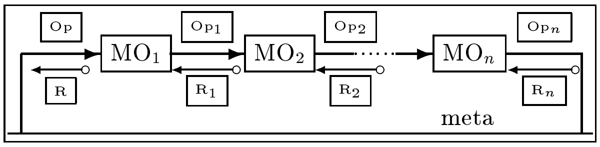
Figure 1: Basic interception.

scheduling operations, to specialized, dissimilar meta-objects. A pre-determined set of aspects can be extended, through intrusive modification of the implementation of the meta-objects responsible for them. We consider this a primitive mechanism of composition, that fails in the general case, because the modifications are very likely to clash.

Several run-time MOPs have been designed so that, when a meta-object is requested to *handle* a reified operation (for example, a method invocation), it is *obliged*, by the design of the MOP, to return a valid result for the operation (typically the value returned by the method), as shown in Figure 1. The meta-level computation that yields the result can include or not the delivery of the operation to the base-level object.

This design implies that the only way to combine the behavior of meta-objects is by arranging for one meta-object, say MO_1, to forward operation handling requests to another, say MO_2, delegating to MO_2 the responsibility for computing the result of the operation. Only after MO_2 returns a result will MO_1 be able to observe and/or to modify it.

Given such a protocol, meta-objects are likely to be organized in a Chain of Responsibility [5, chapter 5], so that each meta-object delegates operation handling requests to its successor, as depicted in Figure 2. The last element of the chain is either the base-level object [9] or a special meta-object that delivers operations to it [16]. We argue that this design presents some serious drawbacks:



Given the basic interception mechanism of Figure 1, meta-objects can only be composed with a Chain of Responsibility [5, chapter 5], a sequential delegation pattern.

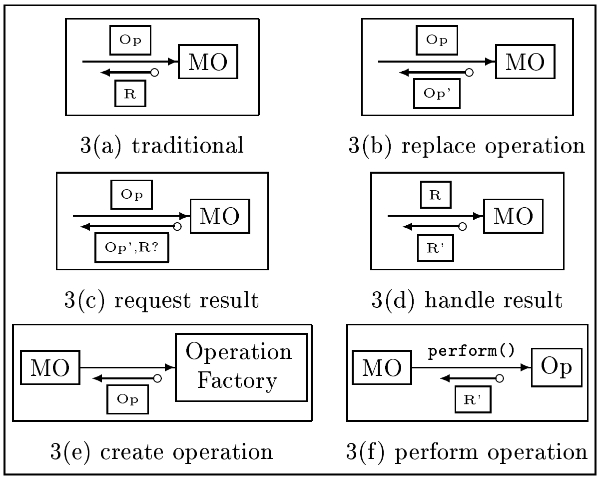
Figure 2: Chain of Meta-Objects.

- it is intrusive upon the meta-object implementation, in the sense that a meta-object must *explicitly* forward operations to its successor;
- it forbids multiple meta-objects from concurrently handling the same operation, because, at a given moment, at most one meta-object can be responsible for producing a result or delivering the operation to the base level;
- it forces meta-objects to receive the results of operations they handled, even if they are not interested in them;
- the order of presentation of results is necessarily the reverse order of the reception of operations, even though different (possibly concurrent) orderings might be more appropriate or efficient, according to the semantics and the requirements of the application;
- it is impossible to mediate interactions between meta-objects and base-level objects with an adaptor capable of resolving conflicts that might arise when multiple meta-objects are put to work together.

Even AspectJ [11, 12], an aspect-oriented programming [8] extension of Java, lacks the possibility of introducing such an adaptor to manage conflicting weaves of aspects so that they can coexist.

3 The Reflective Architecture of Guaraná

The problems presented in the end of Section 2 are solved in the MOP of **Guaraná** by splitting the meta-level processing associated with a base-level operation in the following steps:



*This figure presents the basic MOP of **Guaraná**: although a meta-object is allowed return a result when requested to handle an operation (a), it may prefer to return an operation to be performed (b), with or without an indication that it is interested in its result (c). If it is, it will be presented the result after the execution of the operation (d). Meta-objects can use operation factories to create operations (e) that can replace other operations (b,c) or be performed as stand-alone ones (f).*

Figure 3: Operations and Results.

1. If the target object of the operation is associated with a meta-object, the kernel of **Guaraná**—the entity that implements the MOP— intercepts and reifies the operation and requests the meta-object to handle it; otherwise, no meta-level computation occurs, reducing the overhead for non-reflective objects.

2. A meta-object may produce a result for an operation, as in Figure 3(a). In this case, the meta-level processing terminates by unreifying the result as if it had been produced by the execution of the intercepted operation.

3. However, the meta-object is not required to reply with a result. This permission is essential because *it cannot deliver the operation to the base-level object*. Instead, it should *reply* with an operation to be delivered to the base level (Figure 3(b)) —usually the operation it was requested to handle— and with an indication of whether it is interested in observing and/or modifying the result of the operation (Figure 3(c)).

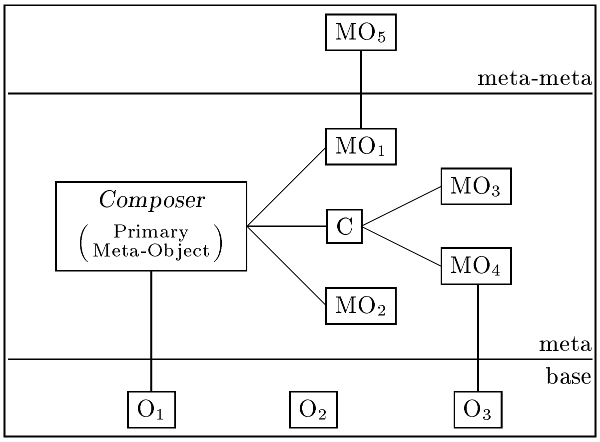
4. Finally, the operation is delivered to the base level, and its result may or may not be presented to the meta-object, depending on its previous reply (Figure 3(d)). If it had requested for permission to modify the result, it may now reply with a different result for the operation.

Replacement operations can be created in the meta-level using *operation factories*, as in Figure 3(e). Operation factories allow meta-objects to obtain privileged access to the base-level objects they manage. Stand-alone operations can also be created with operation factories, and then *performed*, i.e., submitted for interception, meta-level processing and potential delivery for base-level execution, as in Figure 3(f).

We have been able to define *composers* by separating operation handling from result handling, implemented in two distinct methods, namely, handle operation and handle result. A composer is a meta-object that delegates operations and results to multiple meta-objects, then composes their replies in its own replies. For example, a composer can implement the chain of meta-objects presented before, but in a way that one meta-object does not have to keep track of its successor. Another implementation of composer may delegate operations and/or results concurrently to multiple meta-objects, or refrain from delegating an operation to some meta-objects if it is aware they are not interested in that operation.

In **Guaraná**, at any given moment, each object can be directly associated with at most one meta-object, called its *primary meta-object*. If there is no such association, operations addressed to that object are not intercepted, and we say that the object is not reflective at that moment.

The fact that **Guaraná** associates a single (primary) meta-object with an object keeps the design of the interception mechanism simple. Since the primary meta-object can be a composer, as can any meta-object it delegates to, multiple meta-objects can reflect upon an object. These meta-objects form a Composite pattern [5, chapter 4] that we call the *meta-configuration* of that object (Figure 4), a potentially infinite hierarchy of composition that is orthogonal to the well-known infinite tower of meta-objects [13].



The meta-configuration of O_1 is elaborate: a composer, called its primary meta-object, delegates to three other meta-objects, one of which is a composer itself, and delegates to two other meta-objects. O_2 is not associated with any meta-object, so its operations are not intercepted; it is not reflective. O_3 shares MO_4 with O_1. MO_1 is a reflective meta-object, since it has its own (meta-)meta-configuration.

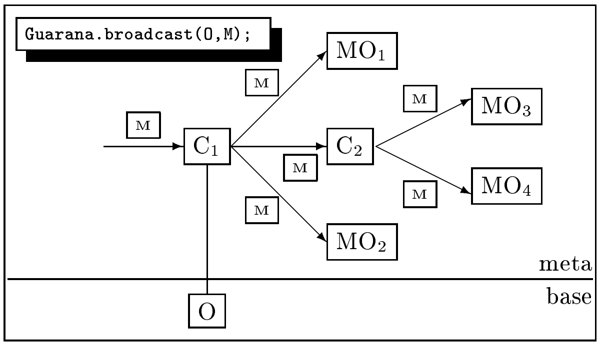
Figure 4: Meta-configurations.

3.1 Meta-configuration management

Guaraná presents two additional features that enforce the separation of concerns between the base level and the meta level: (i) the meta configuration of an object is completely hidden from the base level and even from the meta level itself; and (ii) the initial meta-configuration of an object is determined by the meta-configurations of its creator and of its class, a mechanism we call *meta-configuration propagation*.

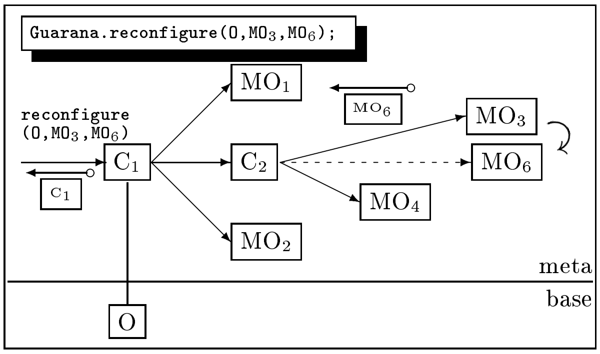
The first design decision implies that there is no way to find out what is the primary meta-object associated with an object. It is possible, however, to send arbitrary *messages* and *reconfiguration requests* to the components of the meta-configuration of an object, through the kernel of **Guaraná**.

Messages can be used to extend the MOP of **Guaraná**, as they allow meta-objects to exchange information even if they do not hold references to each other. Meta-objects that do not understand a message are supposed to ignore it, and composers are expected to forward messages to their components, as in Figure 5. The kernel operation that



Any object M (for message) can be sent to the primary meta-object of an object O. Composers usually forward messages to their components. For non-reflective objects, this request is ignored.

Figure 5: Broadcasting a message.

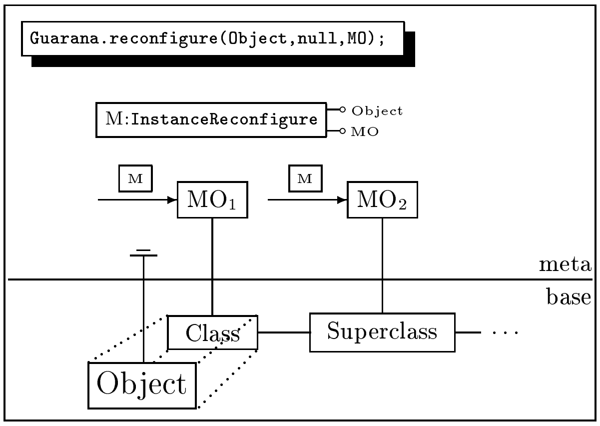


A request to replace MO_3 with MO_6 in the meta-configuration of O was issued. As the request descends the composition hierarchy, it reaches the target meta-object. In this case, it agrees to be replaced, by returning the proposed meta-object. A meta-object must return itself in order to ignore the request, as C_1 does, otherwise the returned meta-object will replace it.

Figure 6: Dynamic reconfiguration.

implements this mechanism is called broadcast.

A reconfiguration request (Figure 6) carries a pair of meta-objects, suggesting that the first meta-object (MO_3) should be replaced with the second (MO_6) in the meta-configuration of object `O`. A special value (`null`) can be used to refer to the primary meta-object. It is up to the existing meta-configuration to decide whether the request is acceptable or not. However, if the base-level object is not reflective, an InstanceReconfigure message is broadcast to the meta-configurations of its class and of its superclasses, as depicted in Figure 7. Their components can modify the suggested meta-configuration, for example, forcing it to remain empty.



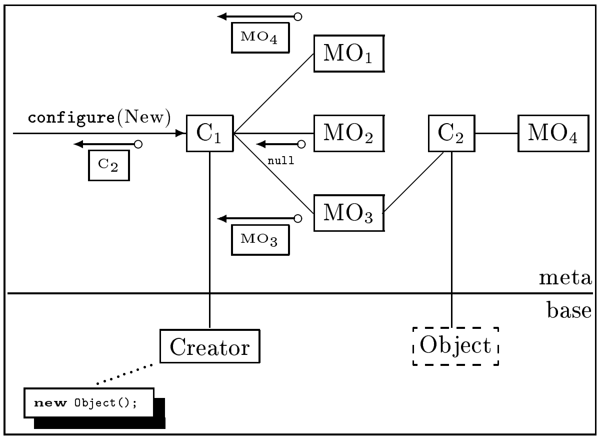
The `null` meta-object can be used as an alias for the primary meta-object in reconfiguration requests. When the object is not reflective, the meta-configuration of its class will be given the opportunity to affect the proposed meta-configuration of the instance. An InstanceReconfigure message will carry the proposed meta-object, so that meta-objects of the object's class(es) may modify it. The remaining meta-object will become the object's primary meta-object.

Figure 7: Reconfiguration of a non-reflective object.

In most object-oriented programming languages, creating an object consists of two steps: (i) allocating storage for the object, possibly initialized with default values, then (ii) invoking its constructor. We say that these steps are performed by the *creator* of the object.

Meta-configuration propagation takes place between these two steps in **Guaraná**. The primary meta-object of the creator is responsible for providing a meta-object for the new object. It may return `null`, a different meta-object or even itself, as a meta-object can belong to multiple meta-configurations. A composer is expected to forward this request to its components and to create a composer that delegates to the meta-objects returned by them, as in Figure 8.

After meta-configuration propagation, the kernel of **Guaraná** broadcasts a NewObject message to the



When a reflective object instantiates another object, its meta-configuration may propagate to the new object before the object is initialized. In fact, the meta-configuration does not have to propagate as a whole: in the picture, only MO_3 was effectively propagated; MO_2 was discarded, whereas MO_1 named MO_4 to occupy its place in the meta-configuration of the new object. C_1 created a new composer to delegate to MO_4 and MO_3.

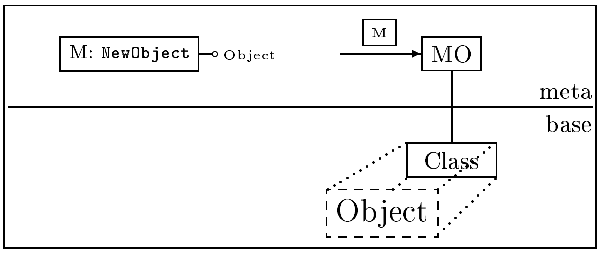
Figure 8: Meta-configuration propagation.

meta-configuration of the class of the new object, so that its meta-objects can try to reconfigure it, as shown in Figure 9. Finally, the object is constructed, but the constructor invocation will be intercepted if the new object has become reflective.

3.2 Support for proxy objects

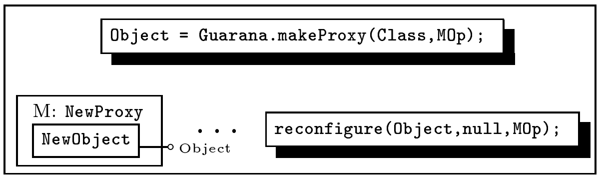
Guaraná provides a mechanism that allows *proxy* objects to be created from the meta level, *without* invoking their constructors. In addition to the traditional use of a proxy, namely, for representing an object from another address space, a proxy can be used to reincarnate an object from persistent storage, to migrate an object, etc.

When a proxy is created, as in Figure 10, the kernel of **Guaraná** broadcasts to the meta-configuration of its class a NewProxy message, a subclass of NewObject. A proxy will usually be given a meta-configuration that prevents operations from reaching it, but it may be transformed in a real object by its meta-configuration, through constructor invocation or direct initialization.



After meta-configuration propagation, the meta-configuration of the class of a new object is notified about the new instance, with a NewObject message, so that it can try to affect the meta-configuration of its instances, by issuing reconfiguration requests.

Figure 9: NewObject messages.



It is possible to request the creation of a proxy object of any class. As soon as the proxy is created, a NewProxy message, subclass of NewObject, is broadcast to the meta-configuration of the class, so that it can take control over the proxy before the proposed meta-object does. Afterwards, a reconfigure request is automatically issued to try to install the proposed meta-object as the primary meta-object of the proxy.

Figure 10: Proxy objects.

3.3 Security

Another advantage of the MOP of **Guaraná** is its concern with security. The hierarchy of composition can be used to limit the ability of a meta-object to affect a base-level object. For example, a composer may decide not to present an operation to a meta-object, or to ignore results or replacement operations it produces. The composer can withhold a message to a component, reject a meta-object produced by a component at a reconfiguration or propagation request, or provide restrictive operation factories to its components, thus limiting their ability to create operations. Furthermore, since the identity of the primary meta-object of an object is not exposed, the hierarchy cannot be subverted.

4 Implementation

We had originally intended to implement **Guaraná** in 100% Pure Java, either by writing an extended Java interpreter in Java or by introducing interception mechanisms through a bytecode preprocessor. The first alternative was discarded because it could imply poor performance and difficulties in handling native methods [22]. A bytecode preprocessor implementation was not possible either, due to restrictions imposed by the Java bytecode verifier [10] and the impossibility to rename native methods, needed in order to ensure their interception.

Therefore, we have decided to implement **Guaraná** by modifying the Kaffe OpenVM^{TM}, an open-source Java Virtual Machine. Most of **Guaraná** is coded in Java, but the Java Virtual Machine has suffered a very minor and localized modification, in order to provide for interception of operations. The performance impact due to the modification was quite small (Section 5) especially when compared to the benefit of transparent interception of method invocations, field and array accesses, object instantiation, and monitor primitives.

The Java Programming Language, however, has not been modified. Thus, any Java program, compiled with any Java compiler, will run on our implementation, within the limitations of the Kaffe OpenVM, the most portable existing Java Virtual Machine. We consider this aspect of **Guaraná** yet another benefit of our approach as programmers will be able to use the reflective mechanisms provided to adapt Java programs originally implemented in the absence of any concern with reflection, even without access to the program's source code. This is possible by starting a meta-application to set up meta-configurations of application classes and objects before the application runs. Then, the meta-application starts the application, but it can still control it through interception, meta-configuration propagation and instance reconfiguration messages. **Guaraná** also provides probe meta-objects that can be helpful for figuring out the behavior of certain objects, so that they can be properly configured.

The MOP of **Guaraná** can also be implemented in other object-oriented programming languages, or even upon existing reflective platforms, as an extension to their built-in MOPs. However, some particular features of **Guaraná** may be difficult to duplicate, if some design decisions for the target language or MOP conflict with those of **Guaraná**.

Java 1.1 was an excellent choice as a target language for **Guaraná**, because it already provides some reflective properties, such as the ability to represent classes, methods and fields as objects (i.e., these elements of the language are reified), so that it is possible to navigate a class hierarchy (introspection) and even interact with objects using the Java Core Reflection API to reflectively invoke methods and to get or set the value of fields. However, such interactions are restricted by the language access control rules, mimicked at run-time. In Java 2, access control can be supressed for particular instances of Methods and Fields, allowing an instance of class that is able to perform the access to supply privileged access to other objects. Other than that, the Reflection API allows an object to perform only the operations that it would have been allowed to perform directly in source code, i.e., access control is based on class permissions.

Guaraná builds upon these features, introducing mechanisms for interception, that are missing in Java, and per-object (as opposed to per-class) security mechanisms, so that meta-objects can obtain privileged access to objects they control.

5 Performance

We have run some performance tests to try to evaluate the impact of introducing reflective capabilities into a Java interpreter. Like the other few papers in the literature on reflection that provide performance data, we have preferred to evaluate the overhead of reflection on each particular operation, instead of running standard benchmarks. In fact, there are no standard benchmarks to evaluate the impact of reflection. Existing general-purpose benchmarks usually focus on optimization of complex patterns of control flow, which would not be affected by the introduction of interception for objects operations, and calculations on large arrays, which would incur a huge overhead.

Our tests have been performed on four different platforms, listed in Table 1. 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 plat-

Table 1: Description of the platforms.

This table describes the platforms on which the performance tests were run.

Tag	Description
i586	100 MHz Pentium running RedHat Linux 5.1
i686	233 MHz Pentium Pro running RedHat Linux 5.0
spu1	167 MHz SPARC Ultra 1 running Solaris 2.6
spu2	200 MHz SPARC Ultra Enterprise 2 running Solaris 2.5

forms, 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.

We have used **Guaraná** 1.4.1 and the snapshot of Kaffe 1.0.b1 distributed with it, using the JIT compiler and the interpreter engines. Kaffe and **Guaraná** were compiled with EGCS 1.1b, with default optimization levels. The program used to perform the tests was the one distributed with **Guaraná** 1.4.1.

For each configuration, we have timed several different operations, described in Table 2. Each operation was timed by running it repeatedly inside a loop, after running it once outside the loop, before starting the timer. This ensures that, before the loop starts, any JIT compilation has already taken place, all the data and code was brought into the cache and, unless the test involves object allocation, the garbage collector will not run.

This inner loop is run repeatedly, with the iteration count being adjusted at every outer iteration, aiming at a running time longer than 1 second. Since the operations that read the clock at the beginning and at the end of each inner loop take less than 1 microsecond to run, and the clock resolution is 1 millisecond, a total running time of 1 second is enough to elliminate any effects they might have in the outcome of the tests.

Table 2: Description of the tests.

This table describes the operation(s) performed within a loop in our performance tests.

Operation	Description
`emptyloop`	No reflective operation.
`synchronized`	Empty block `synchronized` on an arbitrary object.
`invokestatic`	Invoke an empty `static` method that takes no arguments and returns void.
`invokespecial`	Invoke a non-`static` `private` do-nothing method that returns void and takes only the implicit `this` as argument. The same bytecode is used to invoke constructors and, in some cases, `final` methods.
`invokevirtual`	Invoke an empty method that takes only the implicit `this` as argument, and returns void. Dynamic binding, performed with a dispatch table, occurs before interception test.
`invokeinterface`	Invoke the same method, but through an object reference of `interface` type. Dynamic binding is much slower in this case.
`getstatic`	Load a `static int` field into a variable.
`putstatic`	Store a zero-valued variable in a `static int` field.
`getfield`	Load a non-`static int` field into a variable.
`putfield`	Store a zero-valued variable in a non-`static int` field.
`arraylength`	Load the length of an array of `int` into a variable.
`iaload`	Load the first element of an array of `int` into a variable.
`iastore`	Store a zero-initialized variable in the first element of an array of `int`.
`println`	Print the line ``` ``Hello world!'' ``` to System.err, which was redirected to `/dev/null` before starting the Virtual Machine. It is a first attempt to estimate the overall impact of introducing interception abilities.
`compile`	Compile the test program itself. Section 5.1 contains a detailed description and analysis.

Table 3: Overhead on interpreter.

No interception occurs in these tests, they just measure the overhead imposed on the interpreter to introduce the ability *to intercept operations.*

Operation	`i586`	`i686`	`spu1`	`spu2`
emptyloop	−41%	−15%	−0%	−0%
synchronized	−0%	+1%	+0%	+4%
invokestatic	+13%	+0%	+4%	−8%
invokespecial	+30%	+8%	+38%	−10%
invokevirtual	+17%	−0%	+7%	−9%
invokeinterface	−3%	−7%	+20%	−10%
getstatic	−3%	−2%	+20%	−0%
putstatic	−23%	−3%	+24%	+4%
getfield	−22%	−2%	+19%	−0%
putfield	−26%	−2%	+25%	+6%
arraylength	−18%	−9%	+2%	+12%
iaload	−64%	−6%	+1%	−0%
iastore	−14%	−3%	+1%	+1%
println	+6%	+4%	+3%	−2%
compile	+5%	+2%	−2%	−3%

Table 4: Overhead on JIT compiler.

No interception occurs in these tests, they just measure the overhead imposed on the JIT compiler and the code it produces to introduce the ability *to intercept operations.*

Operation	`i586`	`i686`	`spu1`	`spu2`
emptyloop	+0%	+1%	+0%	+0%
synchronized	+12%	+10%	+27%	+3%
invokestatic	+91%	+20%	+23%	+34%
invokespecial	+119%	+8%	+19%	+28%
invokevirtual	+30%	+158%	−6%	+0%
invokeinterface	+7%	+2%	+3%	+2%
getstatic	+68%	+148%	+163%	+163%
putstatic	+180%	+97%	+90%	+90%
getfield	+293%	+86%	+149%	+149%
putfield	+103%	+96%	+66%	+66%
arraylength	+258%	+86%	+140%	+150%
iaload	+191%	+98%	+55%	+95%
iastore	+236%	+55%	+41%	+45%
println	+45%	+6%	+5%	+12%
compile	+36%	+42%	+32%	+29%
compile-JIT	+105%	+112%	+81%	+54%
compile-diff	+16%	+17%	+20%	+20%

The inner-loop iteration count starts at 1, and is repeatedly multiplied by 10 until it is large enough to be measurable with the clock resolution. As soon as this happens, the elapsed time and the iteration count start to be used to estimate the running-time of an iteration. If the total elapsed time of an execution of the inner loop is longer than one second, the estimate is the final result of the test. Otherwise, it is used to compute the iteration count for the next execution of the inner loop, aiming at a total execution time of 1100 milliseconds.

With the exception of the tests `println` and `compile`, this mechanism selected an iteration count between 50,000 and 100,000,000, for the final execution of the inner loop of each test. In the case of `println`, the iteration count was never smaller than 500. The `compile` test was run stand-alone, not within this framework.

Each test case was run 50 times on each configuration and platform, and the average times of the runs were used to compute the relative overheads presented in Table 3 and Table 4. Although we have introduced the ability to intercept operations, no actual interception took place during those tests.

Table 5: Total `compile` time.

These are the total execution times of the `compile` test for each configuration. They were used to calculate the lines `compile` in Table 3 and Table 4.
(times are in seconds)

Configuration	`i586`	`i686`	`spu1`	`spu2`
Kaffe JIT	17	5.1	9.1	7.5
Guaraná JIT	23	7.2	12	9.6
Kaffe interpreter	30	9.2	13	11
Guaraná interpreter	32	9.4	13	10

5.1 The `compile` test

As an additional effort to measure the performance impact of the introduction of interception ability, we have measured the execution time for the Java compiler to translate the test program to Java bytecodes. The averaged execution times are presented in Table 5.

On short-running applications like this, most of the time is spent on virtual machine initialization and JIT compilation, not on running the applica-

Table 6: JIT compilation time for `compile` test.

These are the times spent on JIT compilation during the execution of the `compile` test. They were used to compute the values in the `compile-JIT` line of Table 4.
(times are in seconds)

Configuration	i586	i686	spu1	spu2
Kaffe JIT	3.9	1.3	1.8	1.9
Guaraná JIT	8.0	2.8	3.3	2.9

Table 7: Net `compile` time

These are the differences between total execution time (`compile`) and JIT compilation time (`compile-JIT`), i.e., the times spent on execution of the JIT compiled code. They were used to compute the values in the `compile-diff` line of Table 4.
(times are in seconds)

Configuration	i586	i686	spu1	spu2
Kaffe JIT	13	3.8	7.3	5.5
Guaraná JIT	16	4.5	8.8	6.7

tion itself. The virtual machine start-up, for example, involves executing very large array initialization methods, whose JIT-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 6 shows that more than 50% of the total time was spent on JIT-compiling Java Core classes and the Java compiler itself. Therefore, the actual overhead in execution time, at least for long-running applications, is much smaller.

Table 7 presents the differences between the total time and the JIT-compilation time, that represents the time spent on running the actual application, i.e., the compiler. Long running applications, that repeatedly execute the same methods, should present a reflection overhead similar to the relative overhead of this table.

Table 8: Interception time, interpreter.

*This table presents the interception time of various operations in the **Guaraná** interpreter, with a do-nothing meta-object. Field operations refers to `static` and non-`static` field reads and writes. Array operations involve array length reads and array elements reads and writes.*
(times are in milli*seconds)*

Configuration	i586	i686	spu1	spu2
synchronized	0.92	0.30	0.42	0.35
invokestatic	0.59	0.17	0.22	0.18
invokespecial	0.65	0.17	0.23	0.19
invokevirtual	0.65	0.17	0.24	0.19
invokeinterface	0.67	0.18	0.24	0.19
field operations	0.60	0.16	0.21	0.17
array operations	0.56	0.15	0.21	0.17

Table 9: Interception time, JIT compiler.

*This table presents the interception time of various operations in the **Guaraná** JIT compiler, with a do-nothing meta-object. Other operations refers to all field and array operations.*
(times are in milli*seconds)*

Configuration	i586	i686	spu1	spu2
synchronized	0.55	0.018	0.33	0.25
invokestatic	0.30	0.099	0.20	0.15
invokespecial	0.32	0.10	0.18	0.14
invokevirtual	0.33	0.11	0.20	0.15
invokeinterface	0.33	0.11	0.19	0.15
other operations	0.3	0.09	0.17	0.13

5.2 Intercepting operations

We have also performed some tests involving actual interception, using a do-nothing meta-object to intercept the operation that is the subject of each test. The absolute time spent on the interception of a single operation is presented in Table 8, for the interpreter, and in Table 9, for the JIT compiler.

It is worth noting that each `synchronized` block involves two operations, one that enters the monitor of an object and another that leaves it. Since both are intercepted, the interception time is increased. Additional details are available elsewhere [18].

5.3 Overall discussion

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 results is mostly automated, so there is little 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 machine code on both. It looks like these tests introduce pathological pipeline stalls or branch prediction errors that degrade 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, since these instructions can usually be executed in one or two machine-level instructions, and in **Guaraná** they require at least one more register and two instructions to test for the presence of a meta-object. 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.

It is worth noting that, although we have introduced the ability to intercept object creation, we have not been able to measure the effect of this addition, due to the impredictability of the garbage collector. Anyway, the overhead is known to be negligible, since a single test was introduced in a rather complex function coded in C.

6 Future optimizations

The reflection overhead on the interpreter is quite small. Furthermore, the interpreter is much slower than the JIT compiler, so there is not much point in trying to optimize it any further. For the JIT code, there is little hope for similarly small overheads, though.

One approach we had considered would be to implement all operations, even field and array ones, as invocations of dynamically generated JIT-compiled code. Then, instead of having to test the meta-object reference before performing an operation, an extended dispatch table would contain pointers to these JIT-generated functions, on non-reflective objects, or to interceptor functions, in the case of reflective objects.

However, 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, as in a virtual method invocation, and the absolute time for a virtual method invocation is much larger 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 and array operations are currently implemented. In fact, because of Kaffe's inability to carry register allocation information across basic blocks, the fact that **Guaraná** introduces basic blocks in field or array operations forces registers to be stored in stack slots because it *might* be necessary to invoke an interceptor function. A promising optimization involves improving the register allocation mechanism so as to propagate register allocation information along the most frequently used control flow, that is the one without interception, and move the burden of spilling and reloading registers into the not-so-common case in which interception must take place. 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 with regard to global register allocation, the additional branches introduced by **Guaraná** will not get it confused.

There is another optimization, that is much harder to implement within Kaffe, but that could reduce the overhead of loops and methods that make heavy access of a particular object or array. The test for the existence of a meta-object could be performed before entering the loop or starting the sequence, and different versions of the code would be generated: one, in which no meta-object test is performed for that object, and another in which the test is performed in every iteration, because the meta-object may change. This optimization is based on a similar proposal for optimizing array reference checking [15]. Unfortunately, this kind of optimization can only be performed if no method invocation nor interception could possibly occur within the loop or sequence, so as to ensure that reconfiguration does not take place within the same thread. Even in this case, other threads might reconfigure the object or array while the code runs, so synchronization operations must also be ruled out, because, by definition of the Java Virtual Machine Specification [10], they flush a local cache a thread might maintain. But it may still be worth the effort for array and field operations, given that the overhead imposed on them is still large.

7 Conclusions

Our research on computational reflection was initially motivated by our willingness to verify the use of MOPs as a tool for structuring and building environments for fault-tolerant distributed programming. We intended to design and implement a library like **MOLDS** [19], a library of *reusable* and *combinable* meta-level components useful for distributed applications, such as persistence, distribution, replication and atomicity.

Unfortunately, none of the existing reflective architectures supported composition of meta-objects in a way that fulfilled our needs. Therefore, we started the development of **Guaraná**. This paper is an effort to convey the positive and negative aspects of this experience.

Guaraná provides a powerful and secure mechanism to combine meta-objects into dynamically modifiable, elaborate meta-configurations. In addition to enforcing a clear separation between the reflective levels of an application, the MOP of **Guaraná** improves reuse of meta-level code by defining a meta-object interface that eases flexible composition. Furthermore, it suggests a separation of concernts between meta-objects, that implement meta-level behavior, from composers, that define policies of composition and organization.

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.

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.

Now that we have **Guaraná**, we are concentrating our efforts on the design and implementation of **MOLDS**. The interaction of the various mechanisms foreseen for **MOLDS** will fully demonstrate the power of our MOP. Meanwhile, other projects based on **Guaraná** are demonstrating its flexibility and ease of use. Tropyc [1] is a pattern language for the domain of cryptography, that is currently using **Guaraná** in order to transparently introduce cryptographic mechanisms in electronic commerce applications. The composition strategy of **Guaraná** has also supported the implementation of the Reflective State Pattern and of its adaptation to the domain of fault tolerance [3, 4].

A last evidence of the usefulness of our approach is the possibility of creating a reflective ORB by simply running a 100% Pure Java ORB in **Guaraná**. By doing this, we provide to the users of the ORB the ability to create reflective middleware and applications, with a development cost close to zero.

The experience with the design and implementation of **Guaraná** and related applications allows us to conclude that initiatives by the software industry to build software that is highly adaptable and reusable should incorporate MOPs as flexible as, and at least as efficient as the one we have described.

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*). Additional support is provided by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*), for the PRONEX programme and for a PhD scholarship for Alexandre Oliva.

Islene Calciolari Garcia has helped us very much in reviewing and improving this paper. She also made important contributions to the architecture of **Guaraná**.

Douglas C. Schmidt, from Washington University, St. Louis, has provided us with very useful suggestions for the final version of this paper.

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

References

[1] Alexandre Melo Braga, Cecília Mary Fischer Rubira, and Ricardo Dahab. A system of patterns to cryptographic object-oriented software. In *Pattern Languages of Programs Conference - PLOP'98*, July 1998. TR#WUCS-9825.

[2] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA'95*, volume 30, pages 285–299, October 1995.

[3] Luciane Lamour Ferreira and Cecília Mary Fischer Rubira. Reflective design patterns to implement fault tolerance. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 81–85, Vancouver, BC, Canada, October 1998.

[4] A Reflective Object-Oriented Framework for Developing Dependable Software based on Patterns and Metapatterns. Delano medeiros beder and cecília mary fischer rubira and ricardo dahab. In *28th International Symposium on Fault-Tolerant Computing (FastAbstract)*, June 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch (Designer). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[6] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[7] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97, LNCS 1241*, Finland, June 1997. Springer-Verlag.

[9] Jürgen Kleinöder and Michael Golm. MetaJava: An efficient run-time meta architecture for Java. In *International Workshop on Object Orientation in Operating Systems – IWOOOS'96*, Seattle, Washington, October 1996. IEEE.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Java Series. Addison–Wesley, January 1997.

[11] Cristina Videira Lopes and Gregor Kiczales. *Aspect-Oriented Programming with AspectJ.* Xerox PARC. http://www.parc.xerox.com/aop/aspectj/tutorial.

[12] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ? In *ECOOP'98 Workshop Reader, LNCS 1543*. Springer-Verlag, 1998.

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

[14] Jeff McAffer. Meta-level programming with CodA. In *ECOOP'95*, pages 190–214, August 1995.

[15] Samuel P. Midkiff, José E. Moreira, and Marc Snir. Optimizing array reference checking in java programs. Technical Report 21184 (94652), IBM, T.J. Watson Research Division, Yorktown Heights, New York, June 1998.

[16] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA'95*, volume 30 of *ACM SIGPLAN Notices*, pages 316–330, Austin, TX, October 1995.

[17] Alexandre Oliva and Luiz Eduardo Buzato. Composition of meta-objects in Guaraná. In *Workshop on Reflective Programming in C++ and Java, OOPSLA'98*, pages 86–90, Vancouver, BC, Canada, October 1998.

[18] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical Report IC-98-32, Instituto de Computação, Universidade Estadual de Campinas, September 1998.

[19] Alexandre Oliva and Luiz Eduardo Buzato. An overview of MOLDS: A Meta-Object Library for Distributed Systems. In *Segundo Workshop em Sistemas Distribuídos*, Curitiba, PR, Brazil, June 1998.

[20] 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.

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

[22] Antari Taivalsaari. Implementing a Java Virtual Machine in the Java Programming Language. Technical Report SMLI TR-98-64, Sun Microsystems Laboratories, March 1998.

[23] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 414–434, October 1992.