Efficient Runtime Metaprogramming Services for Java

Article *in* Journal of Systems and Software · April 2019

DOI: 10.1016/j.jss.2019.04.030

CITATIONS
8

READS
150

3 authors:

Ignacio Lagartos
Telefónica, S.A
3 PUBLICATIONS **11** CITATIONS
SEE PROFILE

Jose Manuel Redondo
University of Oviedo
103 PUBLICATIONS **249** CITATIONS
SEE PROFILE

Francisco Ortin
University of Oviedo
126 PUBLICATIONS **732** CITATIONS
SEE PROFILE

Efficient Runtime Metaprogramming Services for Java

Ignacio Lagartos[a], Jose Mauel Redondo[a], Francisco Ortin[a,b,*]

[a] University of Oviedo, Computer Science Department, c/Federico Garcia Lorca 18, 33007, Oviedo, Spain

[b] Cork Institute of Technology, Computer Science Department, Rossa Avenue, Bishopstown, T12 P928, Cork, Ireland

[*] Corresponding author

Notice: This is the authors' version of a work accepted for publication in *Journal of Systems and Software*. Please, cite this document as:

Ignacio Lagartos, Jose M. Redondo, Francisco Ortin. Efficient Runtime Metaprogramming Services for Java. Journal of Systems and Software, volume 153, pp. 220-337, 2019, doi: 10.1016/j.jss.2019.04.030

Efficient Runtime Metaprogramming Services for Java

Ignacio Lagartos[a], Jose Manuel Redondo[a], Francisco Ortin[a,*]

[a] *University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain*

Abstract

The Java programming language and platform provide many optimizations to execute statically typed code efficiently. Although Java has gradually incorporated more dynamic features across its versions, it does not provide several metaprogramming features supported by most dynamic languages, such as structural intercession (the ability to dynamically modify the structure of classes) and dynamic code generation. Therefore, we propose a method to add those metaprogramming features to Java in order to increase its runtime adaptiveness, while taking advantage of the robustness of its static type system and the performance of its virtual machine. We support the dynamic addition, deletion and replacement of class methods and fields, and dynamic code generation. The metaprogramming services are provided as a library, so neither the Java language nor its virtual machine are modified. We evaluate our system, called JMPLib, and compare it with the existing metaprogramming systems for the Java platform and other highly optimized dynamic languages. JMPLib obtains similar runtime performance to the existing fastest system that modifies the implementation of the Java virtual machine. Moreover, our system introduces no performance penalty when metaprogramming is not used, and consumes fewer memory resources than the rest of implementations for the Java platform.

Keywords: Metaprogramming, structural intercession, static type checking, dynamic code evaluation, code instrumentation, runtime performance

1. Introduction

Dynamic languages provide a high degree of runtime adaptiveness, supporting features such as dynamic meta-programming, reflection, dynamic code evaluation, structural intercession, and dynamic reconfiguration and distribution. These features have made dynamic languages appropriate for specific scenarios such as Web development,

*Corresponding author

Email addresses: UO196684@uniovi.es (Ignacio Lagartos), redondojose@uniovi.es (Jose Manuel Redondo), ortin@uniovi.es (Francisco Ortin)

URL: http://www.di.uniovi.es/~ortin (Francisco Ortin)

rapid prototyping, developing systems that interact with data that change unpredictably, and dynamic aspect-oriented programming [1].

For example, in the Web development scenario, Ruby [2] is used for the rapid development of database-backed Web applications with the Ruby on Rails framework [3]. This framework confirms the simplicity of implementing the DRY (Do not Repeat Yourself) [4] and the Convention over Configuration [3] principles using the metaprogramming services of dynamic languages. Nowadays, JavaScript [5] is being widely employed to create interactive Web applications with different frameworks such as Angular [6], Backbone [7] and Ember [8], while PHP is one of the most popular languages for developing Web-based views. Python [9] is used for many different purposes; two well-known examples are the Zope application server [10] (a framework for building content management systems, intranets, and custom applications) and the Django Web application framework [11].

The flexibility of dynamic languages is, however, counteracted by limitations derived from the lack of static type checking. This deficiency implies two major drawbacks: no early detection of type errors, and fewer opportunities for compiler optimizations. Static typing offers the programmer the detection of type errors at compile time, making it possible to fix them immediately rather than discovering them at runtime—when the programmer's efforts might be aimed at some other task, or even after the program has been deployed. Moreover, since the runtime adaptability of dynamic languages is mostly implemented with dynamic type systems, runtime type inspection and type checking commonly involve a significant performance penalty [12].

The Java programming language provides a mature solution to implement efficient and robust object-oriented applications. Its static type system detects many type errors before program execution. Moreover, its highly optimized hotspot virtual machine provides sophisticated optimizations to efficiently run Java applications [13].

Due to the success of dynamic languages, the Java statically typed language has gradually incorporated more dynamic features into their platforms. Java 1.1 included reflection services to examine the structures of objects and classes at runtime (i.e., introspection). These services also allow creating objects and invoking methods of types discovered at runtime. The dynamic proxy class API, added to Java 3, allows defining classes that implement an interface and funneling all the method calls of that interface to an `InvocationHandler`. The Java `instrument` package (Java 5) includes Java *agents* to instrument programs running on the Java Virtual Machine (JVM). The Java Scripting API, added to Java 6, permits scripting programs to be executed from, and have access to, the Java platform. Java 7 adds a new `invokedynamic` instruction to the virtual machine. This new opcode is mainly aimed at simplifying the implementation of compilers and runtime systems for dynamically typed languages [14].

The main contribution of this article is the design and implementation of a lightweight Java library, called JMPLib, to support common metaprogramming services, which can be used with any standard JVM implementation. JMPLib enforces the Java type rules when the application is adapted, providing earlier type error detection. The proposed system supports thread-safe addition, deletion and modification of `static` and instance

methods and fields, interfaces, packages, annotations and generic types, plus the typical functionalities of programmatic dynamic code evaluation. JMPLib provides all these features with similar runtime performance to the most efficient implementation based on the modification of the JVM.

The rest of this paper is structured as follows. Next section describes the background of our research, and the related work is discussed in Section 3. In Section 4, we illustrate JMPLib with a motivating example, and we detail its design and architecture in Section 5. Section 6 depicts an evaluation of our system compared to the related work. The conclusions and future work are presented in Section 7.

2. Background

Java is both an object-oriented language and a platform. The Java platform includes the execution engine (i.e., Java Virtual Machine or JVM) and a set of libraries. There are different Java platforms for different application domains: micro edition for devices with limited resources; standard edition, for general purpose use; and enterprise edition, for multi-tier enterprise applications.

The JVM specification formally describes the features that any JVM standard implementation must support [15]. Although there exist many JVM implementations such as Jikes RVM, Azul Zing JVM and Excelsior JET, the reference one is the Oracle's Java HotSpot Virtual Machine [16]. The HotSpot JVM implementation includes a garbage collector to remove unused objects, a just-in-time (JIT) compiler to generate optimized code at runtime, a set of class loaders, a bytecode verifier, the runtime data areas (e.g., heap memory, methods, threads, registers, etc.), and the native method interface [17, 18].

Inside the Java HotSpot virtual machine, there are actually two separate JIT compiler modes, which are known as C1 and C2 [19]. The C1 compiler generates code quickly and gathers profiling information at runtime. Once the application is properly warmed up (e.g., server-side applications), the C2 compiler recompiles the "hot" methods (those methods consume more execution time). The C2 compiler implements more aggressive optimizations, takes longer to compile and, commonly, provides better runtime performance. In this way, the HotSpot JIT compiler combines the benefits of fast warming up and high performance optimizations for the methods consuming most of the execution time [19].

2.1. Metaprogramming and reflection

Metaprogramming is the ability of computer applications to treat programs as data [20]. Reflection is one of the key techniques used to support metaprogramming. Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [21]. In a reflective language, the computational domain is enhanced with its self-representation, offering its structure and semantics as computable data.

Considering what can be done with the self-representation of applications, reflection can be classified as [22]:

– Introspection: Self-representation of programs can be dynamically consulted but not modified. The Java platform offers introspection by means of the `java.lang.reflect` package. The programmer can obtain information about classes, objects, methods and fields at runtime.

– Intercession: The ability of a program to modify its own execution state, including the customization of its own interpretation or meaning. Adding and removing object fields and methods at runtime is a typical example of intercession provided by dynamic languages such as Python and Ruby. Java does not support intercession.

Another criterion to categorize reflective systems is considering what can be reflected. According to this condition, two levels of reflection are identified:

– Structural reflection: The information reflected is the structure of the program, offered to the programmer as data. In case the program structure is modified, changes will be reflected at runtime. The capability of reading class structures (e.g., Java) and modifying them (e.g., Python) are two examples of structural reflection.

– Behavioral reflection: The ability to access system semantics. In case the semantics is modified, it will involve a customization of the runtime behavior of programs. For instance, MetaXa [23] is a Java extension that allows modifying the method dispatching mechanism at runtime. Meta-Object Protocols (MOPs) are a common way to implement this level of reflection (see Section 3).

Java only provides structural intercession, but it has included different features to improve its metaprogramming capabilities. Java 5 provides *agents* to instrument programs running on the JVM [24]. Java agents allow the transformation of Java classes at load time and the runtime replacement of already loaded classes. The (re)transformation of classes may change method bodies, but it cannot add, remove or rename fields and methods, change their signatures or the inheritance tree (i.e., Java agents do not provide structural intercession).

Java 7 added the `invokedynamic` opcode to the JVM in order to provide a user-defined dynamic linkage mechanism, postponing type checking of method invocation until runtime [25]. `invokedynamic` allows changing the method linked to a call-site and relinking it to another method [26]. Once the link is established, the virtual machine performs its common optimizations, providing better runtime performance than reflection [27].

3. Related work

We analyze the work related to the one presented in this paper. Firstly, we present the systems that provide metaprogramming features for the Java platform. Then, we analyze those that support similar characteristics for other platforms and languages.

3.1. Modifications of the Java platform to support metaprogramming

There are different projects aimed at adding structural intercession to Java. Unlike our system, many of them are based on modifying the implementation of the JVM. The Dynamic Code Evolution Virtual Machine (DCE VM) modifies the JVM to allow the addition and deletion of class members, and the modification of class hierarchies at runtime [28]. DCE VM ensures the type rules of the Java programming language, and also verifies the correct state of program execution. After the adaptation, runtime performance is penalized by 15%, but that value converges to 3% when the JVM reaches a steady state [29]. This is the current reference implementation of the HotSwap functionality originally defined in JSR 292, which was not finally included in the standard platform [25].

JVOLVE modifies the JVM to support evolving Java applications for fixing bugs and adding new features [30]. JVOLVE allows adding, deleting and replacing fields and methods anywhere within the class hierarchy. It modifies the class loader, JIT compiler and garbage collector of the JVM to provide those services. To adapt the running applications, JVOLVE stops program execution in a safe point and then performs the update. Class adaptation is controlled by *transformer* functions that can be customized by the user.

Dynamic Software Updating (DSU) systems migrate a running application to a new version without stopping it. Javelus is a Java HotSpot VM-based DSU system designed to minimize the required pausing time when applications are updated [31]. Instead of performing all the updates at once, Javelus only updates the changed code during the suspension, and migrates stale objects on-demand after the application is resumed, following a lazy approach. Javelus compares the old and new class versions in an application and generates a dynamic patch. This patch contains a summary of class changes, as well as a set of object transformers that modify the objects from the old to the new version. Once a dynamic patch is ready, the programmer can initiate a dynamic update request. Developers can write transformation procedures if the default ones are inappropriate. Updates can only happen when the VM reaches a safe execution point [31].

Iguana/J extends the JVM to provide behavioral reflection at runtime [32]. The programmer can intercept a few Java operations such as object creation, method invocation and field access. The new behavior is specified by the user using and a Meta-Object Protocol (MOP). When a MOP is associated to an object, it handles the operations against that object and provides the services to adapt its execution. Each modifiable operation is represented with one MOP class that the programmer has to extend to define the expected runtime adaptations. The MOP classes and objects are compiled following the Java type system rules.

Java Distributed Runtime Update Management System (JDRUMS) is a client-server system that allows modifying and adding more information to a running program [33]. Servers provide the update services to the clients, which run in the JDRUMS virtual machine. That virtual machine is a JVM extension that provides distributed dynamic updates [34]. Those updates modify the existing classes, distributed as a deployment

kit. For each updated class, a new version is created. Every time an instance of an old version is used, a new instance of the new version is created, its state is transferred to the new object, and the reference is updated. Object migration is controlled by a class that is included in the deployment kit.

In [35], a dynamic classes-enabled virtual machine, called DVM, is proposed. DVM is a modification of the JVM implementation that supports the dynamic adaptation of class structures. A new `ClassLoader` allows loading the modified classes, replacing the existing ones (a functionality not included in the standard JVM). The instances of the adapted classes can evolve in three different ways: no instance is modified, some of them are (depending on user-defined criteria), and all of them are adapted.

3.2. Metaprogramming over the standard Java platform

There exist other works that provide runtime adaptability for Java, implemented as frameworks instead of modifying the JVM. Pukall *et al.* propose unanticipated runtime adaptation, that is adapting running programs depending on unpredictable requirements [36]. They propose a system based on class wrappers and two roles: *caller* (service clients) and *callee* (service providers). A callee is a class wrapper that provides runtime adaptation and access to the original class, implemented with the `instrument` Java 5 package. The callers are aimed at replacing invocations to an object with invocations to the appropriate callee wrapper.

DUSC (Dynamic Updating through Swapping of Classes) is a technique based on the use of proxy classes, requiring no modification of the runtime system [37]. Java agents (i.e., the `instrument` package) is the main Java technology used to change method implementations at runtime. DUSC performs the static modification of classes to allow its later adaptation (making them *swapping-enabled*). They allow adding and deleting classes, but modified ones must maintain their interface (`private` methods and fields can be modified). Another noteworthy limitation is that non-public fields cannot be accessed from outside the class.

Rubah is another framework for the dynamic adaptation of Java applications [38]. When a new dynamic update is available, Rubah loads the new versions of added or changed classes at runtime and performs a full garbage collection (GC) of the program to modify the running instances. The JVM is not modified. Instead, Rubah implements an application-level GC traversal using reflection and some class-level rewriting. To update an application with Rubah, the programmer has to specify the update points, write the control flow migration, and detail the program state migration.

JRebel is a tool to skip the time-consuming build and redeploy steps in the Java development process, allowing programmers to see the result of code changes instantly, without stopping application execution [39]. Modified classes are recompiled and reloaded in the running application. JRebel allows changes in the structure of classes. Classes are instrumented with a native Java agent using the JVM Tool Interface, and a particular class loader. Each class is changed to a master class and different anonymous classes that are dynamically JIT compiled [40]. JRebel does not check that the whole

application has no type errors. Thus, application execution crashes when changes in a class imply errors in a program (e.g., a method is removed and it is invoked elsewhere).

Javeleon allows dynamic updates of Java applications requiring neither JVM modifications nor language extensions [41]. It implements the so-called "in-place proxyfication" technique that instruments application classes with additional bytecode to locate and use the different class and object versions. With the added code, Javeleon can switch forth to an updated class when an invocation to a new method is performed, switching back to the original one if the invocation returns a non-evolved object. The runtime performance penalty of Javeleon is reported to be around 90% in a typical usage, both before and after updates [41]. This penalty is mainly caused by the indirection mechanism added to each class to support the dynamic updates. An extension of this work indicates how to modify the Java virtual machine to directly support dynamic updates, so that the performance of the techniques used by Javaleon could be improved [42].

MetaXa, formerly called MetaJava, is an extension of the Java platform with a reflective meta-level architecture [23]. Structural and behavioral intercession is provided by means of a Meta-Object Protocols (MOP) [43]. By implementing metaobjects, MetaXa permits the programmer to modify the semantics of a few programming language operations, such as object creation, method invocation and field access. MetaXa allows the adaptation of single objects, implementing a class versioning approach with shadow classes. A shadow class is a new particular type for the modified object [23].

3.3. Other approaches to structural intercession

MetaML is a statically typed programming language that supports program manipulation [44]. It allows the programmer to construct, combine and execute code fragments in a type safe manner. In this way, dynamically evaluated programs do not produce type errors. MetaML does not support the manipulation of dynamically evaluated code; i.e., evaluation of code represented as a string, unknown at compile time. Therefore, its metaprogramming features cannot be used to adapt applications to new requirements emerged after their execution.

Fickle$_{\mathrm{II}}$ is a small class-based language that supports the modification of object types at runtime to demonstrate how that feature could be introduced in an imperative, statically typed, class-based, object-oriented language [45]. They define a type change primitive as dynamic object reclassification: a programming language feature that allows an object to change its class membership at runtime while retaining its identity. In Fickle$_{\mathrm{II}}$, a class definition may be preceded by the keywords `root` or `state`. Class reclassification can only occur within a hierarchy rooted with a `root` class. `state` classes are subclasses of `root` classes and they are the only ones that can be reclassified. Classes that are neither `root` nor `state` are respected by reclassification [46]. The Fickle$_{\mathrm{II}}$ implementation of object reclassification offers an advantage over similar approaches (such as wide classes [47, 48]): Fickle$_{\mathrm{II}}$ is type-safe, i.e., any type-correct program (in terms of the type system) is guaranteed to never attempt to access nonexisting fields or methods [49].

Predicate classes are a linguistic construct proposed by Chambers [50] aimed at providing transparent type change functionality at runtime, based on dynamically evaluated predicates. Like a normal class, a predicate class has a set of superclasses, methods and fields. However, unlike a normal class, an object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. That predicate expression can test the state of the object, thereby providing a form of implicit property-based dynamic inheritance (i.e., type change). This linguistic construct was added to the Cecil programming language, which can optionally use static type checking [51]. Because Cecil is prototype-based rather than class-based, the adaptation of predicate classes to Cecil's object model was renamed to predicate objects.

Dynamic C++ classes allow the runtime update of running C++ classes [52]. The existing classes can be modified with different class versions, and other new classes can also be introduced. Each new implementation of a class either updates an existing one (a new version) or introduces a new class (a new type) that implements an interface known at compile time. The instances of the modified classes are not updated. All the new objects are created with the new class version, but the existing ones maintain its version at the moment of creation.

ЯRotor is a modification of the .NET virtual machine to provide native support of structural intercession [53]. ЯRotor extends the class-based object-oriented model with prototype-based semantics to support metaprogramming for both kinds of object models [12]. By using this hybrid object model, it allows modifying the structure of both classes and objects. The JIT compiler is modified to provide structural intercession natively. Therefore, ЯRotor provides important runtime performance improvements compared to similar approaches [22]. Additionally, dynamic inheritance primitives such as changing the class inheritance tree and the type of instances were implemented for both object models [54].

4. Motivating example

This section illustrates part of the metaprogramming services provided by our system, called JMPLib (Java MetaProgramming Library)—the source code is available for download at [55]. In this motivating example, JMPLib is used to programmatically manipulate the structure of a simple `Calculator` class providing two basic methods: `add` and `subtract`. These methods receive two `double` parameters, and return their addition or subtraction.

The code in Figure 1 adds a `lastResult` field (line 6) and a `getLastResult` method (line 15) to `Calculator`. This new field stores the value of the last operation. The code also replaces the existing implementations of `add` (line 8) and `subtract` (line 13) to store in `lastResult` the value of the last operation. Code manipulation can be performed with the modification of the source code as text, and operating on its Abstract Syntax Tree (AST). In this way, line 8 replaces the implementation of `add` by indicating the text of its source code. On the other hand, `sub` is modified by: taking its AST (line 10);

creating an AST that assigns the subtraction of the two parameters to `lastResult` (line 11); adding that new assignment at the beginning of the method body (line 12); and replacing the existing method implementation (line 13).

```
1: public static void adaptationExample(String methodName, double param1, double param2) {
2:   Calculator calc = new Calculator();
3:   // Transaction with different modifications
4:   IIntercessor intercessor = new TransactionalIntercessor().createIntercessor();
5:   // Adds one "lastResult" field
6:   intercessor.addField(Calculator.class,
         new jmplib.reflect.Field(Modifier.PRIVATE, double.class, "lastResult"));
7:   // Changes the implementation of "add" to consider "lastResult"
8:   intercessor.replaceImplementation(Calculator.class, new jmplib.reflect.Method("add",
         // Method signature: return type followed by parameter types
         MethodType.methodType(double.class, double.class, double.class),
         "this.lastResult = a + b; return this.lastResult;"));
9:   // Also changes "sub", but modifying the existing AST
10:  MethodDeclaration md = Introspector.decorateClass(Calculator.class)
         .getMethod("sub", double.class, double.class).getMethodDeclaration();
11:  ExpressionStmt stmt = new ExpressionStmt(new AssignExpr(
         new NameExpr("lastResult"),
         new BinaryExpr(new NameExpr(md.getParameters().get(0).getId().getName()),
               new NameExpr(md.getParameters().get(1).getId().getName()),
               BinaryExpr.Operator.minus)
         , AssignExpr.Operator.assign));
12:  md.getBody().getStmts().add(0, stmt);
13:  intercessor.replaceImplementation(Calculator.class, new Method(md));
14:  // Adds getLastResult
15:  intercessor.addMethod(Calculator.class, new Method("getLastResult",
         MethodType.methodType(double.class), "return this.lastResult;"));
16:  // Executes all the changes at once
17:  intercessor.commit();
18:  // Calls the new "sub" method
19:  calc.sub(3.3, 4.4);
20:  // Gets a Java 8 standard functional interface to invoke getLastResult
21:  IEvaluator evaluator = new SimpleEvaluator().createEvaluator();
22:  Function<Calculator, Double> getLastResult = evaluator.getMethodInvoker(
         Calculator.class, "getLastResult",
         new MemberInvokerData<Function>(Function.class, Calculator.class,
               double.class));
23:  double result = getLastResult.apply(calc);
24:  // Dynamic code evaluation (i.e., eval) to call one method in calc with 2 params
25:  result = evaluator.generateEvalInvoker("calculator." + methodName + "(p1, p2)",
         new EvalInvokerData<>(CalcFunc.class, new String[]{"calculator", "p1", "p2"}),
         ).apply(calc, param1, param2);
26: }
```

Figure 1: Example of metaprogramming using JMPLib.

JMPLib allows the execution of metaprogramming operations both sequentially and using transactions. With transactions, a sequence of operations can be performed at the same time, improving runtime performance. Moreover, if JMPLib is run with multiple threads, transactions are synchronized with the rest of concurrent operations, locking all the modified classes when the transaction is committed—JMPLib is thread safe.

Both sequential and transactional operations are provided with the `IIntercessor` interface. Line 4 shows how to create a new transaction; lines 6, 8, 13 and 15 add

different operations to the transaction; and line 17 executes the transaction by calling the `commit` method. All the changes specified in the operations are type-checked together, similarly to recompiling a new version of the original program. If a type error exists in any of the operations, the system is rolled back and none of the operations are executed.

The Java type system allows the direct invocation to the new implementations of `add` and `sub` methods, as they already existed in the original code (line 19). However, that is not the case for `getLastResult`. Since `getLastResult` is added at runtime, the static type system of Java does not allow the programmer to write code invoking the new method, because `getLastResult` did not exist when the application was compiled. To perform that invocation in a type-safe and efficient way, JMPLib uses the standard functional interfaces included in Java 8 [56]. In line 22 of Figure 1, a `Function` reference is defined to get the new `getLastResult` method, specifying its name, class and parameters. The use of the Java functional interface to call the method (line 23) provides high runtime performance (Section 6), since reflection is not used and no type conversions are performed at runtime.

The type rules of Java are checked when JMPLib is used. For example, an overloaded implementation of `add` could be dynamically added to `Calculator`, only if the number or types of the arguments are changed. If we try to include another `add` method without changing its signature, JMPLib throws a `CompilationFailed` exception with the message *"method add(double, double) is already defined in class Calculator"*. Likewise, if we try to delete a method or field used by another class, JMPLib will throw the same exception at adaptation time to ensure that the new program version follows the Java type rules. This shows that JMPLib not only supports dynamic metaprogramming, but also benefits from the robustness of static type checking.

JMPLib also allows the evaluation of dynamically generated code (line 25), similar to the `eval` function implemented by Lisp, Python and JavaScript. The code to be evaluated can be represented as a `string` or an AST, and dynamically compiled to Java bytecode before its evaluation. In our example, we invoke a method unknown at compile-time (line 25). Its name is passed as the `methodName` parameter to the code in Figure 1. Type safety is ensured with the use of the functional `CalcFunc` interface, which represents the methods in `Calculator` receiving two `double`s and returning another one. The `EvalInvokerData` object is used to pass parameters from the application environment to the code to be evaluated dynamically.

5. Design of JMPLib

JMPLib supports many metaprogramming features provided by most dynamic languages. Those operations include thread-safe structural intercession and dynamic code evaluation (a detailed description of all the primitives provided by JMPLib can be consulted in [57]):

– Addition, replacement and deletion of methods and fields. If a field is replaced, the new one must be a subtype of the existing one. Otherwise, a compiler error is

produced at runtime. The value of the field is kept (or promoted) when its type is replaced.

Likewise, a method could be replaced when its type (i.e., signature) is a subtype of the existing one, according to the Java type system. In other words, signature modification allows contravariant parameters (types of new parameters must be subtypes of the former ones) and covariant return types (a super-type of the original one). This makes the existing invocations to be valid for the new signature.

Methods and fields could be deleted if the rest of the code in the application is not using them. Otherwise, a `CompilationFailedException` is thrown.

- Addition and deletion of interfaces from the interface implementation list (Java `implements`) in a given class or interface.
- Replacement of method implementations. This operation substitutes the implementation of a method, maintaining its signature.
- Dynamic code evaluation. At runtime, JMPLib can execute strings and Abstract Syntax Tree (AST) structures as Java code, returning the result of the code evaluated. Java 8 functional interfaces are used to perform efficient type-safe invocations.
- Addition of new types to running Java applications. This primitive is used to add new classes, interfaces and enumerations without stopping program execution. It can be done by manipulating either text as source code or AST structures to represent the new types.
- Addition and deletion of annotations to classes, interfaces, methods and fields.
- Inclusion of `import` statements to any code to be evaluated at runtime. Without this service, the programmer would need to state the full name (i.e., include its package) of every type used in the new code to be evaluated at runtime.
- Addition and deletion of generic types to classes and methods.

5.1. Architecture

JMPLib manipulates Java programs to provide the runtime metaprogramming services described above. Using the standard elements of the Java platform, our system provides metaprogramming facilities for any platform implementation, with no language extension. The architecture of JMPLib is presented in Figure 2. We first describe the objective of each module; then, we detail the most important ones.

JMPLib implements two Java agents [24]: one for single-threaded applications and another one for concurrent program adaptation[1]. We use the ability of Java agents to replace Java bytecode at load-time and at runtime. At load-time, JMPLib instruments the Java classes to allow its later adaptation (Instrumentation module). At runtime, when a class is adapted (Intercession module), a new version is created and the implementation of the old one is replaced to forward field accesses and method invocations to the new version.

The Intercession module provides the façade to class adaptation. Every time a class is modified, a new class version is generated and loaded into memory. The Class Versions Tables module stores several data structures to locate the different class versions and their members. In this way, JMPLib knows how to modify the implementation of old members, redirecting them to the corresponding ones in the last version.

As shown in Figure 1, the programmer may add new code by manipulating either strings as source code or data structures as ASTs. Additionally, JMPLib provides the programmer the up-to-date source code of the last version in both representations[2] (text and AST). In order to provide that functionality, we included the JavaParser library [58] in JMPLib and integrated it with our Introspection module.

As JMPLib creates new classes and members (e.g., *invokers* and *creators*) to simulate class adaptation, the standard introspection package (`java.lang.reflect`) will reflect those artificial elements at runtime. In order to provide the programmer the abstraction of class adaptation without modifying the Java introspection package, JMPLib implements the Introspection module (`jmplib.reflect`). This new package re-implements the whole `reflect` Java package to provide introspection considering the last class versions. Therefore, it gives the programmer the abstraction that classes are actually modified.

The Introspection module follows the *Decorator* design pattern [59] to integrate JavaParser in JMPLib. As shown in line 10 of Figure 1, `Introspector` decorates any Java element (class, interface, method, etc.) to represent its last version and obtain its source code and AST. The types provided (`Class`, `Method`, `Field`, etc.) are used to interact with the Intercession and Dynamic Code Evaluation modules (Figure 2).

JMPLib generates Java code at runtime and compiles it with the Java compiler (Java 6 `ToolProvider`) [60], generating the `.class` files to be loaded into memory. Type rules are dynamically checked by the Java compiler. A `CompilationFailed` exception containing the compiler output is thrown if any error exists. Unlike most dynamic languages, the error is produced when the code is compiled—at adaptation time—, not when an invalid statement (e.g., invoking a deleted method) is executed—which could happen potentially much later than the adaptation.

The Dynamic Code Evaluation module provides services to evaluate code created at runtime as text or as AST structures. The `eval` method allows the evaluation of Java

[1]By default, the agent for single-threaded applications is used; otherwise, the `thread_safety` parameter must be passed to the agent.

[2]JMPLib also provides the source code of the Java API, if it has been installed in the JDK.

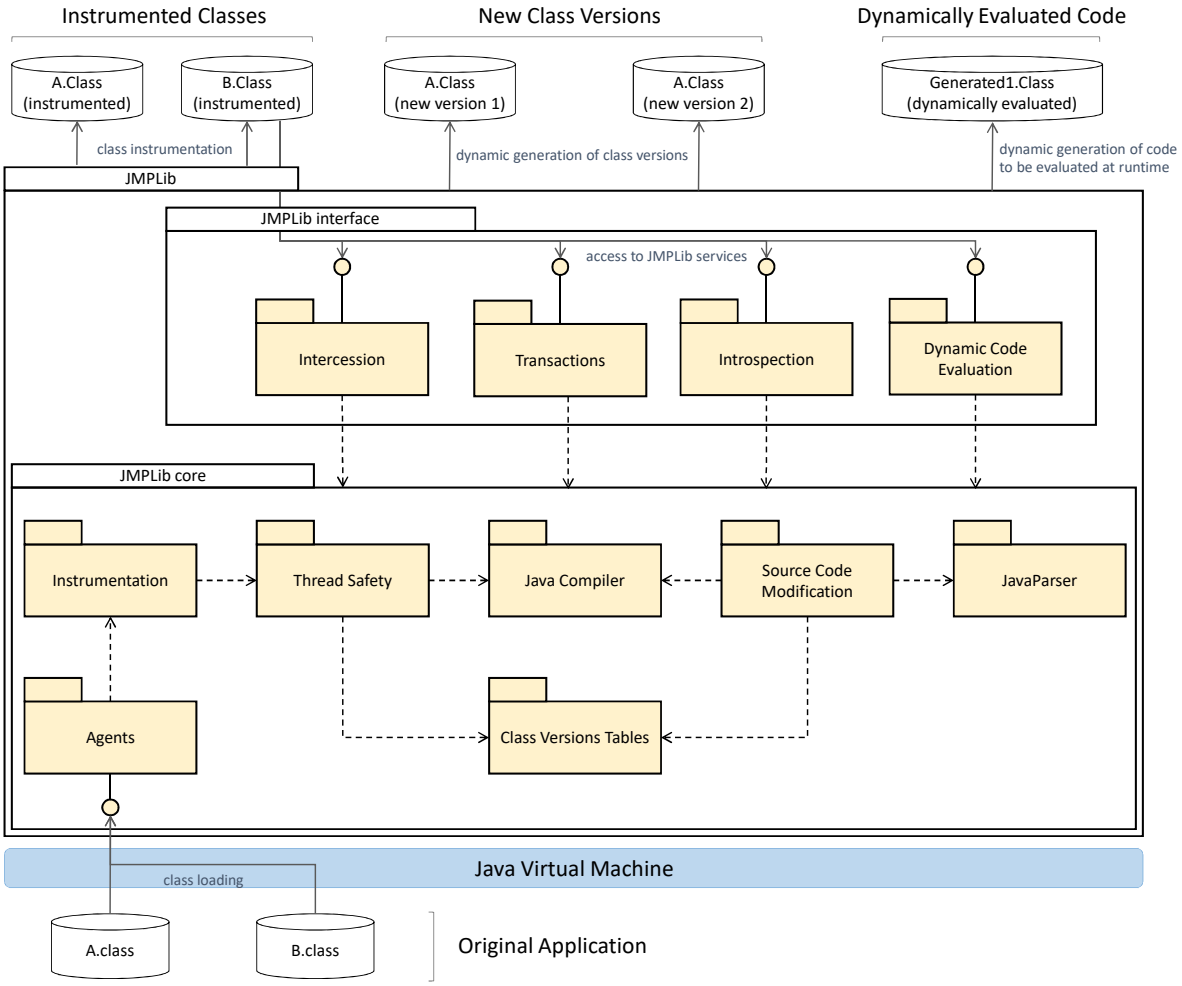

Figure 2: Architecture of JMPLib.

expressions. After the dynamic compilation process, a type-safe reference pointing to the `eval` method is returned to the caller. The use of functional interfaces allows us to prevent the use of reflection and type casts, obtaining significant runtime performance benefits (Section 6.3). The `exec` method generates and loads into memory a new type at runtime.

When programmers write new code to be evaluated at runtime, they are not aware of the different class versions. Therefore, they will use the members in the last class version, but writing code for the original one. In order to avoid the compiler errors produced by this difference, their code must be transparently converted into another one that considers the existing class adaptations. For instance, code invoking an added method in an existing object will crash at runtime, since the new method belongs to the last class version, not to the object class (a previous version). The Source Code Modification module performs syntax and semantic analysis (detailed in Section 5.4) to achieve the type-based modification of the code to be evaluated, making sure that it

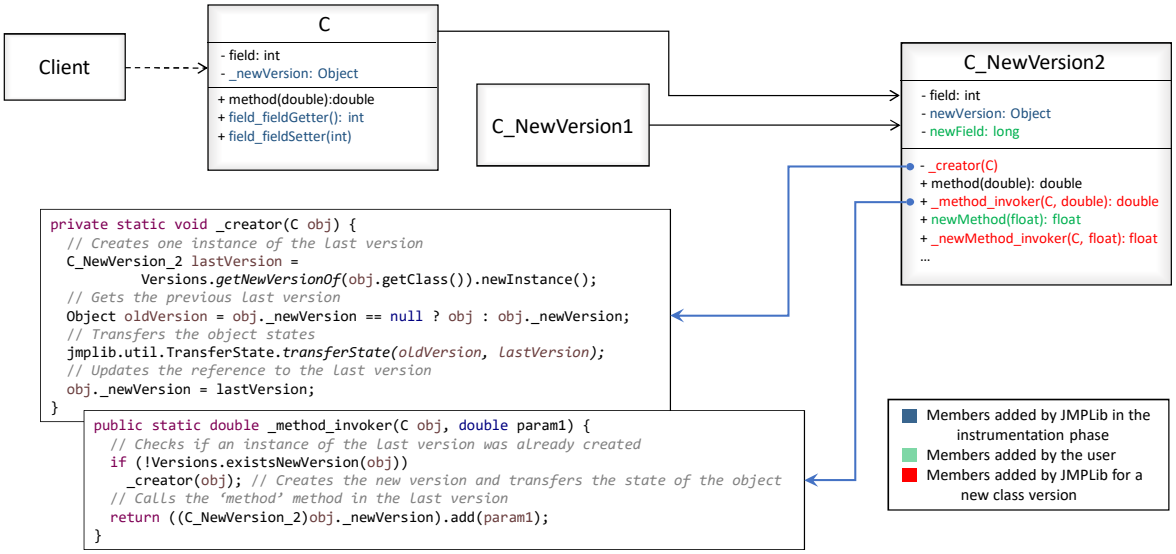

Figure 3: Class versions in memory after class adaptation.

uses the last versions in the program.

JMPLib allows the concurrent adaptation of programs. The Thread Safety module provides this capability when the `thread_safety` parameter is passed to the Java agent. In that case, JMPLib generates thread-safe code when classes are instrumented and new versions are dynamically generated. Different monitors are used to synchronize the adaptation of classes with the runtime system that forwards method invocations and field accesses, and transfers object states from one class version to another.

The Transaction module ensures that all the adaptations in a transaction are executed atomically. If an error exists, it rolls back all the modifications and none occurs. For concurrent programs, no other adaptation could be performed when a transaction is committed. Moreover, a transaction only creates a new version of the adapted classes regardless the number of modifications, providing better runtime performance.

5.2. Instrumentation

Instrumentation takes place when the JVM loads each class before its execution, modifying the class implementation to support dynamic metaprogramming. We use ASM to instrument the classes directly in their binary form [61]. The instrumentation process consumes CPU time, but it is only performed once per class, and it commonly takes place at the beginning of execution (Section 6.4).

As shown in Figure 3, the instrumentation process adds a `_newVersion` field to every class at load time. This `Object` field will dynamically point to its last updated version, which reflects all the changes performed by the programmer. Our system transparently uses the `_newVersion` field to efficiently transfer the state of objects from old class versions to the last one.

Figure 3 also shows the members added to each class by the Instrumentation module. Besides the `_newVersion` field, a pair of get / set methods for each field are also added

to each class (`fieldGetter` and `fieldSetter`). These two methods are used to forward field access to the last class version through `_newVersion` (next subsection).

The instrumentation process also modifies the bytecode for accessing fields outside their class (i.e., when `this` is not the reference used to access the field). Those field accesses are replaced with invocations to the corresponding get / set methods added. In this way, JMPLib ensures that the accessed field belongs to the last updated version (`_newVersion`). Next subsection explains why this process is not necessary inside the class scope (i.e., when `this` is used), avoiding the performance cost of using the get / set methods.

The Instrumentation module also replaces the occurrences of the `instanceof` and `checkcast` bytecodes. Since class adaptation is implemented with the creation of new classes at runtime, these two opcodes must consider such new classes. Therefore, the instrumentation agents replace the `instanceof` and `checkcast` bytecodes with the corresponding introspective code that checks whether the object is an instance of any of the existing class versions.

In the design of JMPLib, we carefully considered that changes in the instrumentation process must not include significant penalties in runtime performance (Section 6.4). The reason of this design criterion is that all the classes are instrumented and, in real applications, only some of them are adapted.

5.3. Intercession

When a class is modified at runtime, a new class is created holding the new members plus the existing ones. As shown in Figure 3, JMPLib creates the new `C_NewVersion1` and `C_NewVersion2` classes when the `newField` and `newMethod` members are added to `C`. The last version provides the new members (`newMethod` and `newField`) plus the original ones (`method` and `field`).

When the implementation of one method is modified, JMPLib assigns the new body to the new class version, but the rest of classes in the application keep calling the original one. In Figure 3, `C_NewVersion2` holds the new implementation of `method`, but the existing code (e.g., the `Client` class) invokes the original method in `C`. To overcome this problem, the method implementations of earlier class versions (`method` in `C`) are replaced with an invocation to an *invoker* method added to the new version (`_method_invoker` in Figure 3).

JMPLib adds to the new class version one *invoker* method for each original one. The objective of *invoker* methods is twofold. First, they transfer the object state from the original class to the new version. This is only done once per object, so next invocations to any *invoker* method using that object will skip the state transfer operation. In this way, we make sure that object state is transferred before executing any method in the new class version. The second objective of *invoker* methods is to call the last version of the associated method (`method` in `C_NewVersion2`). Therefore, any invocation to `method` in `C` ends up executing the same method in `C_NewVersion2` (after transferring the state of the object).

Figure 3 depicts the code of *invoker* methods. Their signatures include the original type (`C`) as its first parameter, followed by the method arguments. *Invokers* first check whether the object state has already been transferred to the last class version. If so, they just call the last implementation of the associated method. Otherwise, the state is transferred by calling the `_creator` method added by JMPLib to each class version.

The `_creator` method (Figure 3) creates an instance of the last version, and gets the previous version. Then, it transfers the last state to the new instance representing the last version. The `_newVersion` field in the original object is finally updated with the new instance. Therefore, the subsequent method invocations using that object will not need to transfer the object state. Moreover, if a new version is created and the object is not used, the object state is not transferred (it is lazy).

As shown in Figure 3, different versions of a class may be created. All the different versions point to the last one, and all the method implementations in old versions are replaced with invocations to the new corresponding *invoker* method in the last version. The result is that the existing calls to old methods will invoke the last implementation with only one level of indirection (*invoker*), regardless the number of existing class versions.

As mentioned, JMPLib distinguishes two kinds of field accesses. For field access where the implicit object is `this`, the code is left unchanged. This is because JMPLib complies that: 1) methods in old versions are never executed (its body is replaced with invocations to the new version); and 2) object state is transferred to the last class version before executing any method. Therefore, `this` field accesses will always get the last object state, since it is previously transferred.

The second case scenario of field access is when the object is not `this`. In that case, if fields were accessed directly, it would not be ensured that the code would access the last version state. This is the reason why, in the instrumentation phase, JMPLib adds a pair of get / set methods for each field, and replaces this second kind of field accesses with invocations to those methods. In this way, those field accesses are actually treated as methods by JMPLib. Notice that this scenario is not very common, since fields are commonly declared as `private` or `protected`.

When a method or field is deleted, the new class version will not include it. Since Java type checking is performed upon adaptation, JMPLib ensures that the new application version does not perform direct calls to the deleted class member. However, that invocation could still happen using reflection, if the program keeps a `Method` or `Field` reference pointing to the old version. For this particular case scenario, JMPLib replaces the body of the deleted methods in the original class with one statement throwing a `MethodNotFound` exception. Similarly, the bodies of the get / set method pairs for each field throw `FieldNotFound` exceptions.

5.4. Dynamic code evaluation

This module of JMPLib provides primitives to perform dynamic code evaluation of text and AST structures. The `exec` method is used to create a new Java type (class,

interface or enumeration), check that it fulfills the language rules, and load it into memory, returning the new type created (a `Class<?>` object).

The typical `eval` function supported by most dynamic languages is offered by the `generateEvalInvoker` method. As shown in Figure 2, JMPLib dynamically generates a new class (e.g., `Generated1`) with one `eval` method returning the Java expression to be evaluated at runtime. After the dynamic compilation process, a type-safe reference pointing to the `eval` method is returned to the caller. The use Java of functional interfaces allows us to avoid the use of reflection and type casts, obtaining significant runtime performance benefits (Section 6.3).

If the dynamically generated code has any error, JMPLib throws a runtime exception. Otherwise, the generated code is loaded, and hence instrumented with the process described in Section 5.2. Therefore, dynamically generated classes can also be dynamically adapted by JMPLib.

The Dynamic Code Evaluation module is not a mere wrapper of the Java compiler. To provide the programmer the abstraction of runtime adaptable classes, it is necessary to perform different changes in the programmer's code before its dynamic evaluation. For example, the code `"obj.newMethod(value)"` invokes the new method added to `C`, not present in the original version of the class. The invocation uses an instance of the original `C` class (`obj`). However, that new method is not actually added to `C` (it is added to `C_NewVersion2`), so the code written by the user would result in a compiler error. For this reason, JMPLib replaces that code with `"C_NewVersion2._newMethod_invoker(obj, value)"`.

The Source Code Modification module performs all the necessary changes in the code to be evaluated at runtime. These changes are not only made to the code in the `eval` and `exec` primitives, but also to the body of methods added and replaced at runtime. These are the main modifications done by the Source Code Modification module:

1. Replace all the invocations to methods in adapted classes with the corresponding *invoker* in the last version.
2. Substitute `this` and `super` references. Since the dynamically evaluated code is placed in a new class generated at runtime, those two references must be replaced with references pointing to the appropriate objects.
3. Modify the field accesses of adapted objects with invocations to the corresponding get / set methods in the last class version.
4. Modify all the `static` method invocations and field accesses to use the last class version.

To perform these code modifications, we used the Polyglot open source Java compiler [62]. After the semantic analysis phase, we traverse the type-annotated Abstract Syntax Tree (AST) checking for the types inferred by the compiler. Then, the Source Code Modification module consults the Class Version Tables and modifies the code to ensure that it uses the last up-to-date version of every type.

5.5. Thread safety

If the `thread_safety` parameter is passed to the Java agent upon application execution, JMPLib provides all its services (Section 5) in a thread-safe way. Different threads may adapt the application classes concurrently while the application is running, since JMPLib transparently synchronizes all the operations. In thread-safe mode, JMPLib performs the following additional operations:

- Class adaptation is synchronized. Section 5.3 details how, when a class is adapted, a new class version is generated and the code in the old versions are replaced. JMPLib synchronizes the intercession operations on a per-class basis: there cannot be two concurrent adaptations of the same class. For this purpose, we use the `Class<?>` object of the original class as a monitor.

- Object state transfer (`_creator`) and method invocation and field access (`_invoker`) are synchronized. In the instrumentation phase, a `ReadWriteLock` instance field is added to every class. This Java monitor allows multiple simultaneous reader threads on a per-object basis, but exclusive writers. Therefore, for each object, there could be simultaneous *invokers*, but they will be locked by transfers of object states (`_creator`s). This makes the runtime system to be thread safe and efficient, since concurrent method invocations are allowed.

- Method invocations (*invokers*) and class adaptations are exclusive. When an object field is being accessed or one of its method is being invoked, its class cannot be adapted by another thread. As in the first case scenario, we use the object class of the first class version as a monitor. Object state transfer is also synchronized with class adaptation, since `_creator`s are always called by `_invoker`s.

- Transactions. Since transactions may involve the adaptation of multiple classes, the concurrent modification of those classes and the accesses to their members are locked when the transaction is committed. All the monitor objects of the classes adapted are used to synchronize the execution of the transaction.

- Dynamic code evaluation primitives can be executed simultaneously. However, neither JavaParser nor Polyglot are thread safe. Therefore, JMPLib synchronizes the use those tools to generate and transform code. In this way, only one thread could be generating code with each tool.

- In thread-safe mode, JMPLib uses synchronized versions of the data structures in the Class Version Tables module (`ConcurrentHashMap` instead of `HashMap`).

6. Evaluation

This section is aimed at answering the following research question:

Is the proposed design and implementation suitable to provide a wide set of metaprogramming services for the Java platform without modifying the JVM, providing competitive runtime performance and memory consumption?

In order to answer the research question, we first perform a qualitative comparison of the metaprogramming services provided by JMPLib and the most similar works discussed in Section 3. Then, we evaluate the runtime performance and memory consumption of JMPLib, using different benchmarks and real applications. We compare them with different systems and languages that provide similar metaprogramming features.

6.1. Qualitative evaluation

In the qualitative evaluation, we selected the systems in the related work section that are more similar to JMPLib. We classify them as systems providing metaprogramming a) by modifying the JVM (Section 3.1); b) over the standard Java platform (Section 3.2); and c) by using MOPs. The purpose of this comparison is not to identify the best system, but rather to see whether the proposed design and implementation succeeds in supporting most of the features provided by the existing systems (i.e., the first part of the research question).

Table 1 shows the qualitative comparison. Regarding the intercession primitives, most of the systems support the addition, deletion and replacement of methods and fields. DCE VM, Pukall *et al.* and DUSC support neither replacing field types without losing their values, nor replacing method signatures without changing their implementations (Pukall *et al.* only provides those features for `public` and non-`final` members; and DUSC just allows removing `private` members). MetaXa does not support intercession of `static` members. It can also be seen in Table 1 that many systems do not allow the manipulation of annotations, generic types, native methods, interface implementation, and changing the base class (dynamic inheritance). JMPLib supports those features except changing the base class. Although the runtime model of JMPLib was designed to allow this functionality, we do not have a working implementation yet—next version is expected support it. Lastly, only DVM, Pukall *et al.* and the two MOPs allow the adaptation of single objects. In that case, a new class is created for that particular instance (i.e., schema versioning [63]), causing different problems such as maintaining the class data consistency, class identity, usage of class objects in the code, garbage collection, inheritance reliability and memory consumption [64]. Due to all these problems caused by class versioning, we decided not to include this feature in JMPLib.

Table 1 also analyzes the additional features discussed in the article. Around half of the systems, including JMPLib, provide their services using the standard Java platform and language, and they allow dynamic updates in arbitrary execution points (not in particular places). Only JMPLib and DUSC provide transactions. JMPLib and Iguana/J allow calling their services programmatically; other services support those features indirectly (e.g., class loaders or plugins must be implemented) or partially (e.g., the whole

new class must be specified instead of one single member). The structure of the new class versions could be consulted with introspection services in all the systems but DUSC and Iguana/J. MetaXa and JDRUMS are the only systems that are not thread safe.

Only JMPLib and Javeleon enforce Java type rules at adaptation time, since both are based on source code recompilation. However, the Javeleon instrumentation process causes a runtime performance overhead of 89.4% [41], whereas JMPLib adds no instrumentation overhead (see Section 6.4). JVOLVE defines a mechanism to enforce type rules at adaptation time based on bytecode recompilation, but the Jikes RVM used in the implementation does not perform bytecode verification when bytecode is recompiled [30]. DVM allows type rule enforcement when the JIT compiler and method inlining are disabled, causing a 92% runtime performance overhead [35]. Other systems such as JRebel and DCE VM do not check whether member deletion causes type errors if the code in other classes use such deleted members, so applications will throw type error exceptions when the deleted members are accessed.

The last classification of features includes dynamic code evaluation. JMPLib is the only system that supports that kind of metaprogramming features. The rest of the systems are focused on structural intercession.

6.2. Methodology of the quantitative evaluation

This section comprises a description of the systems and benchmark suites used in the evaluation, together with the explanation of how data is measured and analyzed.

6.2.1. Selected systems

We selected different systems that provide the metaprogramming features supported by JMPLib. First, we include those systems similar to ours, aimed at providing metaprogramming services for the Java language:

- Dynamic Code Evolution Virtual Machine (DCE VM) 64-Bit Server (build 25.71-b01-dcevmlight-26, mixed mode). DCE VM is a modification of the JVM that allows the dynamic addition and deletion of class members [28]. DCE VM permits the programmer to replace the loaded classes with new class versions.

- JRebel 7.0.4 [39]. JRebel is a Java agent that allows reloading modified Java classes at runtime, without stopping the application execution. JRebel is commonly used to speed up the development process of Java applications by skipping the time-consuming build and redeploy steps.

- JMPLib 1.1.0. This version of JMPLib provides the metaprogramming services described in this article (Section 5).

- Java 64-bit 1.8.181 for Windows 10. Although Java does not provide structural intercession, we include the execution of the original Java programs with no metaprogramming to compare the reflective systems with the baseline one.

		Modified VMs					Frameworks over JVM						MOPs	
	Feature	**DCE VM**	**JVOLVE**	**Javelus**	**JDRUMS**	**DVM**	**JMPLib**	**Pukall *et al.***	**DUSC**	**Rubah**	**JRebel**	**Javeleon**	**Iguana/J**	**MetaXa**
Structural Intercession	1	Yes	Yes	Yes	Yes	Yes	Yes	Partially	Partially	Yes	Yes	Yes	No	Partially
	2	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Partially
	3	Yes	Yes	Yes	Yes	Yes	Yes	Partially	Partially	Yes	Yes	Yes	No	Partially
	4	Partially	Yes	No	Yes	Yes	Yes	Partially	No	Yes	Yes	Yes	No	Partially
	5	Yes	Yes	Yes	Yes	Yes	Yes	Partially	Yes	Yes	Yes	Yes	Yes	Partially
	6	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	No
	7	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	No
	8	No	No	No	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No
	9	No	No	No	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No
	10	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	Yes
	11	Yes	No	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes	No	No
	12	Yes	No	Yes	No	Yes	Partially	Partially	No	Yes	Yes	Yes	No	Yes
	13	No	No	No	No	Partially	No	Yes	No	No	No	No	Yes	Yes
Additional Features	14	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No
	15	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	16	No	No	No	No	No	Yes	No	Yes	No	No	No	No	No
	17	No	No	No	Partially	Partially	Yes	No	Partially	No	Partially	No	Yes	No
	18	No	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
	19	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes
	20	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
	21	No	Partially	No	No	Partially	Yes	No	No	No	No	Yes	No	No
Code Evaluation	22	No	No	No	No	No	Yes	No	No	No	No	No	No	No
	23	No	No	No	No	No	Yes	No	No	No	No	No	No	No
	24	No	No	No	No	No	Yes	No	No	No	No	No	No	No
	25	No	No	No	No	No	Yes	No	No	No	No	No	No	No
	26	No	No	No	No	No	Yes	No	No	No	No	No	No	No
	27	No	No	No	No	No	Yes	No	No	No	No	No	No	No

Table 1: Qualitative comparison of systems related to JMPLib (the features are 1: add/remove fields, 2: replace field type without losing its value, 3: add/remove methods, 4: replace method signature without changing its implementation, 5: replace method implementation, 6: add/remove/replace annotations from classes, 7: add/remove/replace annotations from methods, 8: add/remove/replace generic types from classes, 9: add/remove/replace generic types from methods, 10: modify native method implementations, 11: add/remove/replace interfaces, 12: add/remove/replace base class, 13: adaptation of single objects, 14: standard virtual machine, 15: standard Java language, 16: combined operations through transactions, 17: programmatic support of metaprogramming primitives, 18: dynamic updates in arbitrary execution points, 19: access to the dynamic structure of adapted classes through introspection services, 20: thread-safety, 21: Java type rules enforcement at adaptation time, 22: dynamic evaluation of expressions, 23: dynamic creation of new types, 24: creation, modification and evaluation of ASTs, 25: source code extraction from reflected metaobjects, 26: ASTs extraction from reflected metaobjects, 27: add/remove/replace of imports for expressions).

Second, to analyze alternative ways to obtain metaprogramming over the Java platform, we include popular languages for the JVM that provide metaprogramming:

– Jython 2.7.0 (formerly called JPython) is a 100% pure Java implementation of the Python programming language, seamlessly integrated with the Java platform [65]. Jython 2.7.0 is compatible with Python 2.7 and supports structural intercession and dynamic code generation.

– Rhino 1.7.1.1 is an open-source implementation of JavaScript written entirely in Java, developed by Mozilla [66]. It can be embedded into Java applications to

provide scripting capabilities. Since Java 6, Rhino is included in the Java standard edition as the default JavaScript engine.

– Groovy 2.5. Groovy is an optionally-typed and dynamic language for the Java platform [67]. It provides many metaprogramming services, and supports type inference and compilation to `.class` files. We compiled all the Groovy programs with `groovyc` before its execution and annotated them with `@CompileStatic` to enable all the compile-time optimizations provided by the language.

Third, we measure other reflective language implementations, not for the JVM, that provide state-of-the-art optimizations, and offer important runtime performance gains [68, 69]:

– PyPy 5.0.1 [70]. PyPy is an alternative Python implementation that provides a tracing JIT compiler to optimize program execution at runtime, generating dynamically optimized machine code for the hot code paths of commonly executed loops. PyPy has been evaluated as the fastest Python implementation for most kinds of applications, including metaprogramming [71].

– V8 6.1.0 is the Google JavaScript engine used in Chrome, which can run both standalone and embedded into C++ applications [72]. V8 implements a runtime adaptive JIT compiler that dynamically optimizes the generated code, based on heuristics of the code execution profile. For the hotspot functions detected at runtime, the JIT compiler applies aggressive optimizations including inline caches, type feedback, customization, control flow graph optimizations and dead code elimination.

– SpiderMonkey 24.2.0 is the JavaScript engine of Mozilla, currently included in the Firefox Web browser and the GNOME 3 desktop [73]. It uses three optimization levels: an interpreter, the baseline JIT-compiler, and the IonMonkey compiler for more powerful optimizations. The baseline compiler generates binary code dynamically, collecting more accurate type information and applying basic optimizations. Finally, IonMonkey is only triggered for hotspot functions, providing optimizations such as type specialization, function inlining, linear-scan register allocation, dead code elimination, and loop-invariant code motion.

We do not include the Iguana/J and MetaXa MOPs because they provide poor runtime performance. Object creation in Iguana/J encounters a 25x delay when the classes are adapted, and the cost of method interception is 24x because it uses reflection [74]. In MetaXa, when a class is modified, method invocation, field access and object creation is 28, 23 and 8.5 times slower than Java [75].

6.2.2. Selected benchmarks

We measure the execution of different applications, from synthetic microbenchmarks to real applications:

- Microbenchmark. In order to evaluate the runtime performance of the metaprogramming primitives provided by JMPLib, we implemented eight different synthetic microbenchmarks that use metaprogramming services in different case scenarios.

- SciMark 2.0, a Java benchmark for scientific and numerical computation [76]. More precisely, fast Fourier transformations (FFT), dense LU matrix factorization, Monte Carlo integration that approximates the value of Pi (MonteCarlo), and Jacobi successive over-relaxation (SOR).

- Shootout. This benchmark is composed of different well-known algorithms implemented in different programming languages [77]. We run those tests that do not perform any I/O interaction, which are: BinaryTrees, Fannkuchredux, Mandelbrot, Nbody and SpectralNorm.

- Real large scale applications taken from the Java Grande benchmark (its third section) [78]:

    - Euler: solves the time-dependent Euler equations for flow in a channel with a bump on one of the walls.
    - MonteCarlo: a financial simulation that uses Monte Carlo techniques to price products derived from the price of an underlying asset.
    - MolDyn: a N-body code that models the behavior of N argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions.
    - Search: solves a game of connect-4 on a 6x7 board using an alpha-beta pruned search technique.
    - RayTracer: measures the performance of a 3D ray tracer. The scene rendered contains 64 spheres, and is rendered at a resolution of 25x25 pixels.

As described in Section 6.2.1, we measure programs coded in Java (JRebel, DCE VM and JMPLib), Python (Jython and PyPy), Groovy and JavaScript (Rhino, V8 and SpiderMonkey). We took Java programs and manually translated them into the other languages. Although this translation might introduce a bias in the runtime performance of the translated programs, we thoroughly checked that the same operations were executed in all the implementations. We verified that the benchmarks compute the same results in all the programs.

Table 2 shows the number of methods, classes, lines of code and different complexity measures of the code used in the evaluation. The data presented in Table 2 is for the original Java application.

		Number of methods	Number of classes	Lines of code	Non-comment LOC	Average cyclomatic complexity	Average design complexity	Average essential complexity	Total cyclomatic complexity
Java Grande	Euler	26	4	5,481	3,089	4.04	2	1.24	101
	MolDyn	17	4	453	402	3.65	1.71	1	62
	JGMontecarlo	167	14	2,947	1,125	1.51	1.25	1.22	252
	Search	26	5	612	411	4.04	1.85	2.15	105
	Raytracer	76	17	906	710	1.62	1.34	1.17	123
SciMark	FFT	32	7	502	409	2.78	1.41	1.34	89
	LU	36	7	596	470	2.89	1.33	1.28	104
	MonteCarlo	25	7	385	304	2.64	1.32	1.32	66
	SOR	27	7	392	331	2.63	1.33	1.3	71
Shootout	BinaryTrees	21	7	220	207	1.81	1.48	1.43	38
	Fannkuchredux	19	6	210	201	2.21	1.47	1.58	42
	Mandelbrot	19	6	198	191	2.05	1.47	1.68	39
	Nbody	28	8	287	285	1.68	1.46	1.25	47
	SpectralNorm	24	7	274	262	1.96	1.62	1.29	47

Table 2: Details of the code used in the evaluation (the cyclomatic complexity of one method measures the number of linearly independent paths through its execution; the design complexity measures how interlinked a method control flow is with calls to other methods; and the essential complexity is a graph-theoretic measure of just how ill-structured a method control flow is [79]).

6.2.3. Data analysis

We follow the methodology proposed by Georges *et al.* [80] to evaluate the runtime performance of applications, including those executed on virtual machines that provide JIT-compilation. In this methodology, two approaches are considered: 1) *start-up* performance is how quickly a system can run a relatively short-running application; 2) *steady-state* performance concerns long-running applications, where JIT compilation does not involve a significant variability in the total running time, and hot-spot dynamic optimizations are applied.

To measure start-up performance, a two-step methodology is used:

1. We measure the execution time of running multiple times the same program. This results in p (we take $p = 30$) measurements x_i with $1 \leq i \leq p$.
2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is calculated using the *Student's* t-distribution because we took $p = 30$ [81].

In the subsequent figures, we show the mean of the confidence interval plus the width of the confidence interval relative to the mean (bar whiskers). If two confidence intervals do not overlap, we can conclude that there is a statistically significant difference with a 95% (1-α) probability [80].

The steady-state methodology comprises the following four steps:

1. Each application (program) is executed p times ($p = 30$), and each execution performs at least k ($k = 10$) different iterations of benchmark invocations, measuring

each invocation separately. We refer x_{ij} as the measurement of the j^{th} benchmark iteration of the i^{th} application execution.

2. For each i invocation of the benchmark, we determine the s_i iteration where steady-state performance is reached. The execution reaches that state when the coefficient of variation (CoV, defined as the standard deviation divided by the mean) of the last k iterations (from s_{i-k+1} to s_i) falls below a threshold (2%). To avoid an influence of the previous benchmark execution, a full heap garbage collection is done before performing every benchmark invocation. Garbage collection may still occur at benchmark execution, and it is included in the measurement. However, this method reduces the non-determinism across multiple invocations due to garbage collection kicking in at different times across different executions.
3. For each application execution, we compute the $\overline{x_i}$ mean of the k benchmark iterations under steady state:

$$\overline{x_i} = \frac{\sum_{j=s_{i-k+1}}^{s_i} x_{ij}}{k}$$

4. Finally, we compute the confidence interval for a given confidence level (95%) across the computed means from the different application invocations using the *Student's* t-statistic described above. The overall mean is computed as $\overline{x} = \sum_{i=1}^{p} \overline{x_i}/p$. The confidence interval is computed over the $\overline{x_i}$ measurements.

6.2.4. Data measurement

To measure the execution time of each benchmark invocation, we instrument the applications with code that registers the value of high-precision time counters provided by the Windows operating system. This instrumentation calls the native function `QueryPerformanceCounter` of the `kernel32.dll` library. This function returns the execution time measured by the Performance and Reliability Monitor of the operating system [82]. We measure the difference between the beginning and the end of each benchmark invocation to obtain the execution time of each benchmark run.

Memory consumption is measured following the same methodology to determine the memory used by the whole process. For that purpose, we use the maximum size of working set memory employed by the process since it was started (the `PeakWorkingSet` property). The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. Those pages are resident and available for an application to be used without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including those from the process modules and the system libraries. The `PeakWorkingSet` was measured with explicit calls to the services of the Windows Management Instrumentation infrastructure [83].

All the tests were carried out on a 3.50 GHz Intel Core i5-4690 system with 8 GB of RAM, running an updated 64-bit version of Windows 10. We used the 64-bit Java

Virtual Machine 1.8.0.144. All the benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded.

If the P_1 and P_2 programs run the same benchmark in T and $2.5 \times T$ milliseconds, respectively, we say that runtime performance of P_1 is 150% (or 2.5 times) higher than P_2, P_1 is 150% (or 2.5 times) faster, P_2 requires 150% (or 2.5 times) more execution time than P_1, or the performance benefit of P_1 compared to P_2 is 150%—the same for memory consumption. To compute average percentages, factors and orders of magnitude, we use the geometric mean.

6.3. Runtime performance

We developed a synthetic microbenchmark to first measure runtime performance of the metaprogramming primitives; afterwards, we evaluate more realistic programs. The microbenchmark measures execution time of `static` and instance method invocations, and `static` and instance field accesses for reading and writing operations; all of them added at runtime with intercession.

Figure 4 displays the execution times of the operations measured relative to JMPLib. Startup and steady-state execution times are the same in this scenario, because each operation is measured in a loop of 10,000 iterations. Therefore, the benchmark execution reaches a steady state, producing the same results for both methodologies.

The average performance cost of JMPLib compared to Java is 17%, the lowest one among the systems providing metaprogramming for the Java language (DCE VM, JRebel and JMPLib). JRebel requires 5% more execution time than JMPLib. DCE VM is only 1% slower than JMPLib, so there is no statistical significant difference between them (i.e., 95% confidence intervals overlap [80]).

The reflective languages for the Java platform are significantly slower than the systems for the Java language: Jython, Rhino and Groovy require, respectively, 12, 23 and 131 times more execution time than JMPLib. Analyzing the three highly-optimized implementations of dynamic languages, they perform notably better than the languages for the JVM. There is no significant difference between PyPy and JMPLib, and SpiderMonkey and V8 require 11% and 119% more average execution time than our system.

The execution times of the microbenchmark give us an initial evaluation of the cost of metaprogramming. However, we should also evaluate more realistic programs where metaprogramming is used in the implementation of real algorithms. To this aim, we took 14 programs from the 3 benchmark suites described in Section 6.2.2. All the programs were implemented using metaprogramming: structural intercession was used to add all the methods and fields to classes; and methods were implemented with dynamic code generation operations. Then, we run the application and measure its execution time with the two methodologies described in Section 6.2.3.

Figure 5 shows the average startup and steady-state execution times for the 14 applications, relative to JMPLib startup (Table 3 details the non-relative steady-state execution times for all the benchmarks). For both methodologies, JMPLib provides the best runtime performance, requiring 23% (startup) and 21% (steady-state) more

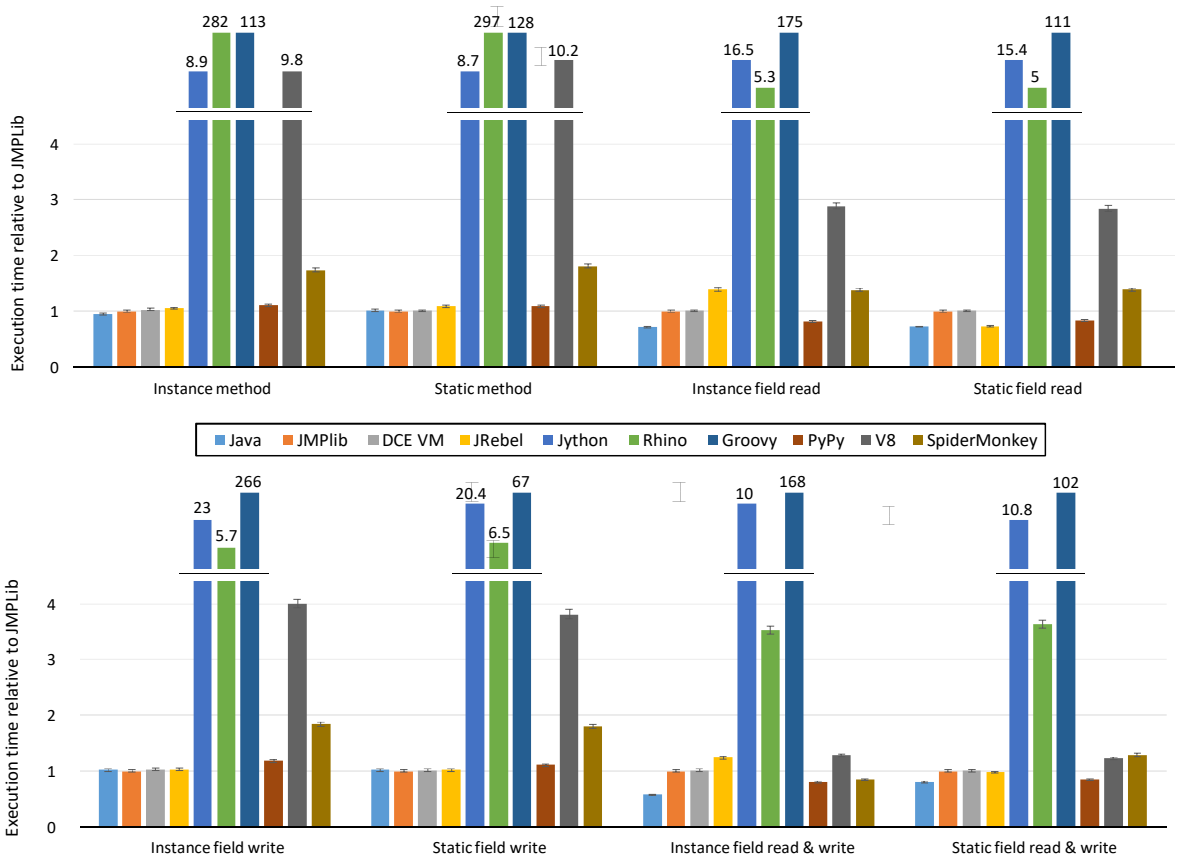

Figure 4: Microbenchmark execution times relative to JMPLib.

execution time than Java with no meta-programming. Again, the difference between DCE VM and JMPLib is no statistically significant (on average, JMPLib is 2% faster than DCE VM). The other system for the Java language, JRebel, is the third fastest implementation, consuming 63% and 68% more CPU time than JMPLib.

The three implementations of dynamic languages for the Java platform, provide a slower metaprogramming alternative to the systems in the first group. Jython, Rhino and Groovy are, respectively, 49, 182 and 94 times slower than JMPLib for startup; and 45, 151 and 84 times for steady state.

The highly-optimized languages provide significantly better results than those compiled for the Java platform. Compared to JMPLib, V8, SpiderMonkey and PyPy require 23%, 50% and 250% more execution time for startup; and 49%, 89% and 327% for steady state. Therefore, the JVM does not seem to be an efficient platform for dynamically typed programs that use metaprogramming.

JMPLib provides metaprogramming services for the Java language taking advantage of the hotspot runtime optimizations implemented by the JVM. These optimizations achieve better runtime performance than the most optimized implementations of languages supporting metaprogramming. Moreover, the proposed system, which can be used over any JVM implementation, provides similar runtime performance to the ex-

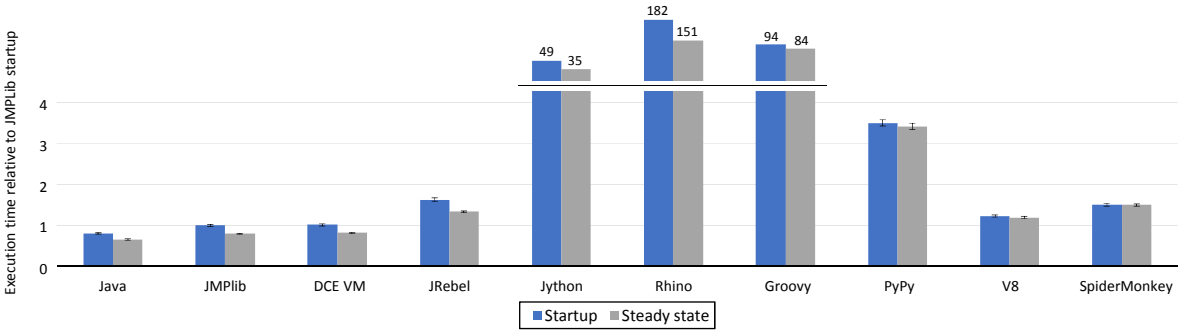

Figure 5: Average startup and steady-state execution times for 14 applications, relative to JMPLib startup.

		Java	JMPLib	DCE VM	JRebel	Jython	Rhino	Groovy	PyPy	V8	Spider Monkey
SciMark	FFT	238	253	261	257	11,133	61,947	10,290	1,687	265	346
	LU	418	576	589	579	127,025	423,181	62,919	5,848	1,973	2,888
	MonteCarlo	76	89	92	373	2,488	16,420	3,466	854	331	481
	SOR	263	355	362	394	12,893	48,432	8,418	1,119	485	481
Shootout	BinaryTrees	37	43	44	66	488	1,931	13,460	48	62	69
	Fannkuchredux	194	189	196	193	4,426	21,429	3,289	301	236	204
	Mandelbrot	671	723	738	737	67,698	259,166	8,438	7,000	1,436	1,498
	Nbody	264	270	276	2,025	25,363	61,338	51,016	1,537	289	478
	SpectralNorm	989	1,024	1,058	1,398	36,072	134,569	306,345	4,010	940	1,393
Java Grande	Raytracer	727	1,795	1,882	4,177	69,935	317,567	587,252	4,746	1,556	2,432
	Euler	2,243	2,537	2,585	20,954	119,988	513,243	1,099,269	13,757	4,145	5,435
	MolDyn	875	941	963	1,075	35,968	157,426	197,499	3,363	1,065	1,639
	JGMontecarlo	1,833	2,487	2,572	2,740	125,222	483,691	582,911	14,614	4,565	4,811
	Search	1,427	1,830	1,884	1,862	88,464	476,210	369,708	9,011	2,666	2,791

Table 3: Steady-state execution times (milliseconds) for the SciMark 2.0, Shootout and Java Grande (large-scale applications) benchmarks. All the programs were implemented using metaprogramming: structural intercession was used to add methods and fields to classes; and methods were implemented with dynamic code generation operations. Then, the execution time of the programs are measured.

isting fastest implementation that modifies the JVM.

All the systems improve their performance for the steady-state methodology. However, all the systems running over the JVM show significantly higher improvements (from 12% to 37%) than the three non-JVM dynamic languages (from 1% to 3%). Furthermore, the Java platform provides better optimizations when the application reaches a steady state at runtime.

6.4. Performance cost of metaprogramming

As described in Section 5.2, JMPLib performs code instrumentation when classes are loaded into memory. To provide dynamic metaprogramming, classes are extended and parts of their code are modified. In this section, we evaluate the cost of that instrumentation, measuring the same programs as in the previous section but without using metaprogramming. Therefore, we evaluate the cost of running a program in a metaprogramming system when no metaprogramming is used.

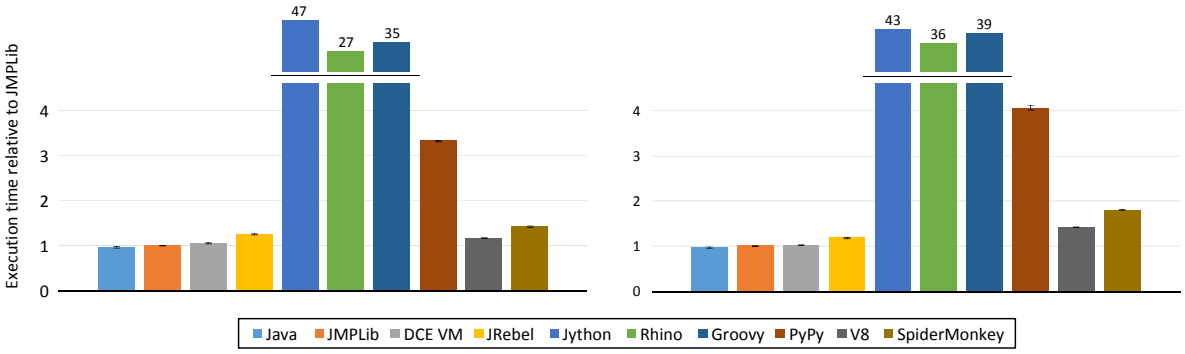

Figure 6: Average startup and steady-state execution times when no metaprogramming is used, relative to JMPLib.

Figure 6 shows average execution times for the 14 programs selected, when no metaprogramming primitive is used. All the execution times are lower than those in the previous section. In fact, there is no statistically significant difference between Java, JMPLib and DCE VM for both startup and steady state. JMPLib is 25% (startup) and 18% (steady state) faster than the other system for Java (JRebel).

The dynamic languages for the JVM are at least 25 times slower than JMPLib, showing that the JVM executes Java much more efficiently than dynamic languages. Although differences with the highly-optimized languages are not as high as those with the dynamic languages for the JVM, they show execution times from 17% to 232% higher than JMPLib. Therefore, the statically typed Java language performs notably better than dynamic languages, even when metaprogramming is not used. Consequently, JMPLib benefits from the performance gains of both the Java language and the JVM.

6.5. Performance cost of thread safety

As mentioned, JMPLib provides a thread-safe execution mode that must be indicated to the Java agent at startup. To evaluate its performance cost, we measure the execution time of all the programs in Sections 6.3 and 6.4 in the thread safe mode.

On average, the programs that perform class adaptations at runtime (Section 6.3) require 22.66% more execution time when the thread safe mode is enabled. If no metaprogramming is used (Section 6.4), the performance difference is lower than the error (2.8%), so there is no statistically significant difference.

6.6. Memory consumption

Figure 7 shows the average memory consumption for the 14 applications, relative to Java. It distinguishes the two scenarios of Sections 6.3 and 6.4: when metaprogramming primitives are used, and when only instrumentation takes place (no metaprogramming).

We can see how JMPLib is the JVM-based system that consumes fewer memory resources. When no metaprogramming is used, it requires 97% more memory than Java, and 4 times more when metaprogramming is used. As shown in Table 4, an

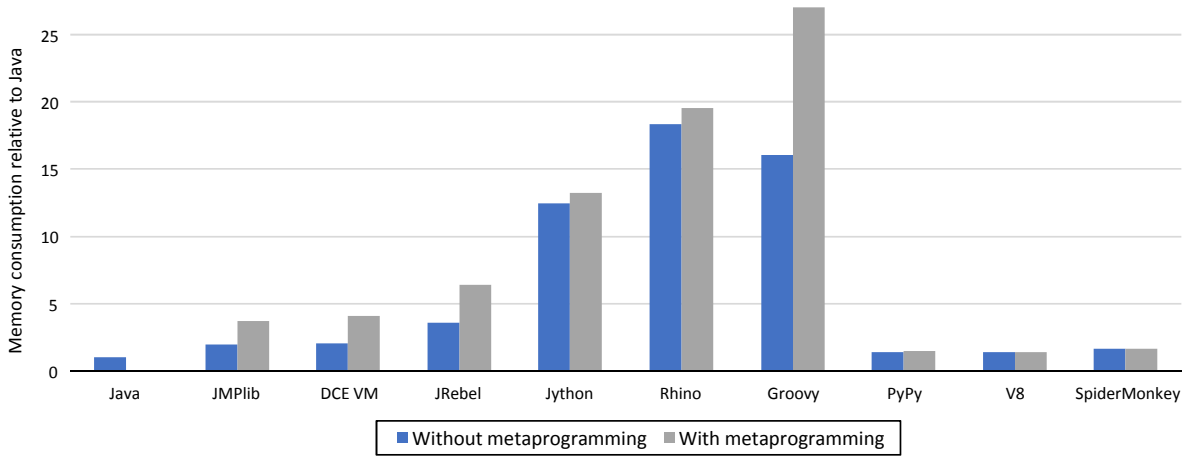

Figure 7: Average memory consumption with and without metaprogramming, relative to Java.

instrumented application in JMPLib loads, on average, 157% more classes than Java, causing the additional memory consumption. Those additional classes represent the implementation of JMPLib plus the extra code added in the instrumentation phase. When metaprogramming is used, Table 4 shows how JMPLib is the system that loads more classes in the JVM to support application adaptation (136% more classes than the instrumentation scenario).

DCE VM consumes 104% more memory than Java if no adaptation is required (Figure 7). In that scenario, it loads 112% more classes than baseline Java program (Table 4), so DCE VM creates a significant number of new types to allow the later adaptation of the application. When metaprogramming is used, DCE VM consumes 4.1 more memory than Java and only loads 7% more classes than the previous scenario (no metaprogramming). Therefore, most of the runtime adaptation is implemented inside the virtual memory.

In Figure 7, we can see that JRebel consumes more memory than the two previous systems: 2.6 and 5.4 times more than Java for instrumentation and metaprogramming scenarios, respectively. Like JMPLib, JRebel is implemented over the standard JVM. However, its implementation requires 182% more classes than JMPLib for instrumented applications, and 23% more for program adaptation.

7. Conclusions

JMPLib shows how structural intercession and dynamic code generation can be included in the Java platform efficiently, without modifying its virtual machine. We instrument the classes at load time, replacing the implementation of methods to support a lazy state transfer mechanism with one single level of indirection. Our system takes advantage of the hotspot dynamic optimizations implemented by the JVM and the robustness of the Java type system to provide good runtime performance and early type error detection. Therefore, it is necessary to modify neither the JVM nor the language implementation.

		Java	JMPLib		DCE VM		JRebel	
		No-meta	No-meta	Meta	No-meta	Meta	No-meta	Meta
Java Grande	Euler	424	1,353	3,067	1,001	1,091	3,494	3,602
	MolDyn	588	1,349	3,072	995	1,102	3,492	3,597
	JGMontecarlo	437	1,375	3,165	1,013	1,101	3,503	3,611
	Search	421	1,353	3,079	1,000	1,095	3,494	3,599
	Raytracer	460	1,208	2,981	1,023	1,027	3,517	3,628
SciMark	FFT	450	1,188	2,891	1,013	1,089	3,518	3,618
	LU	451	1,188	2,860	1,014	1,089	3,524	3,619
	MonteCarlo	451	1,188	2,874	1,014	1,090	3,519	3,624
	SOR	451	1,189	2,864	1,014	1,090	3,520	3,613
Shootout	BinaryTrees	655	1,238	2,894	1,173	1,249	3,599	3,707
	Fannkuchredux	452	1,201	2,866	1,013	1,089	3,505	3,613
	Mandelbrot	452	1,201	2,888	1,013	1,089	3,510	3,614
	Nbody	454	1,203	2,892	1,015	1,091	3,515	3,624
	SpectralNorm	628	1,226	2,865	1,026	1,119	3,516	3,647

Table 4: Classes loaded at runtime by systems that support metaprogramming for Java (*meta*: applications created with metaprogramming, i.e. Section 6.3; *no-meta*: applications not using metaprogramming, i.e., Section 6.4).

The evaluation showed that the proposed system provides similar runtime performance to DCE VM, the fastest existing system, which is implemented as a modification of the JVM. For the 14 applications measured, JMPLib performs significantly better than the metaprogramming languages for the JVM, and even better than the existing highly-optimized non-JVM language implementations. JMPLib provides higher performance benefits when the JVM reaches a steady state at runtime. Our system introduces no runtime performance penalty when metaprogramming is not used, and consumes fewer memory resources than the rest of implementations for the Java platform.

We are currently working on supporting dynamic inheritance (i.e., modification the base class). At present, we allow modifying the base class of any non-native class—if the source code of the Java platform is included in the JDK installation. In our current version, native classes cannot be extended, and overriding of native methods is not supported either.

Currently, JMPLib requires the source code of classes to provide structural intercession. Although the standard library can be adapted (if the source code of the JDK is included upon installation), we plan to overcome this limitation in future versions. ASM will be used to perform code manipulation at the bytecode level.

The last step of the project is to allow the dynamic adaptation of whole Java applications. After modifying, recompiling and testing the new version of a program, it could be replaced at runtime without stopping its execution and maintaining its state [84].

The current release of the JMPLib library, its source code, the execution time and memory consumption tables, and all the examples and benchmarks used in this paper can be downloaded from `http://www.reflection.uniovi.es/invokedynamic/download/2018/jss`

Acknowledgments

This work was funded by the European Union, through the European Regional Development Funds (ERDF), and the Principality of Asturias, through its Science, Technology and Innovation Plan (Grant GRUPIN14-100). The authors also received funds from the Banco Santander through its support to the Campus of International Excellence.

References

[1] F. Ortin, J. M. Cueva, Dynamic adaptation of application aspects, Journal of Systems and Software (2004) 229–243.

[2] D. Thomas, C. Fowler, A. Hunt, Programming Ruby, 2nd Edition, Addison-Wesley, 2004.

[3] D. Thomas, D. H. Hansson, A. Schwarz, T. Fuchs, L. Breed, M. Clark, Agile Web Development with Rails. A Pragmatic Guide, Pragmatic Bookshelf, 2005.

[4] A. Hunt, D. Thomas, The pragmatic programmer: from journeyman to master, Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1999.

[5] ECMA-357, ECMAScript for XML (E4X) Specification, 2nd edition, European Computer Manufacturers Association, Geneva, Switzerland, 2005.

[6] Google, Angular: One framework. Mobile & desktop, `http://angular.io` (2018).

[7] DocumentCloud, Backbone.js, `http://backbonejs.org` (2018).

[8] Tilde inc., Ember: a framework for creating ambitious web applications, `http://emberjs.com` (2018).

[9] G. van Rossum, L. Fred, J. Drake, The Python Language Reference Manual, Network Theory, United Kingdom, 2003.

[10] A. Latteier, M. Pelletier, C. McDonough, P. Sabaini, The Zope book, `http://zope.readthedocs.io/en/latest/zope2book/` (2018).

[11] Django Software Foundation, Django, the web framework for perfectionists with deadlines, `https://www.djangoproject.com` (2018).

[12] F. Ortin, M. A. Labrador, J. M. Redondo, A hybrid class- and prototype-based object model to support language-neutral structural intercession, Information and Software Technology 44 (1) (2014) 199–219.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the Java HotSpot Client Compiler for Java 6, ACM Transactions on Architecture and Code Optimization 5 (1) (2008) 7:1–7:32.

[14] Oracle, Java Virtual Machine Support for Non-Java Languages, http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html (2018).

[15] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, The Java Virtual Machine Specification, Java SE 8 Edition, Addison Wesley, 2014.

[16] Oracle, The Java HotSpot virtual machine – technical white paper, http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf (2018).

[17] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the Java HotSpot client compiler for Java 6, ACM Transactions on Architecture and Code Optimization 5 (1) (2008) 1–32.

[18] B. Venners, Inside the Java Virtual Machine, 1st Edition, McGraw-Hill Professional, 1999.

[19] Z. Majo, T. Hartmann, M. Mohler, T. R. Gross, Integrating profile caching into the hotspot multi-tier compilation system, in: Proceedings of the 14th International Conference on Managed Languages and Runtimes, ManLang 2017, ACM, New York, NY, USA, 2017, pp. 105–118.

[20] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[21] P. Maes, Computational reflection, Ph.D. thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel (January 1987).

[22] F. Ortin, J. M. Redondo, J. B. G. Perez-Schofield, Efficient virtual machine support of runtime structural reflection, Science of Computer Programming 74 (2009) 836–860.

[23] M. Golm, J. Kleinöder, Jumping to the meta level, in: P. Cointe (Ed.), Meta-Level Architectures and Reflection, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 22–39.

[24] Oracle, Java agents to instrument programs running on the JVM, https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html (2018).

[25] Oracle, JSR 292: Supporting Dynamically Typed Languages on the Java™ Platform, https://jcp.org/en/jsr/detail?id=292 (2011).

[26] P. Conde, F. Ortin, JINDY: A Java library to support `invokedynamic`, Computer Science and Information Systems 11 (1) (2014) 47–68.

[27] F. Ortin, P. Conde, D. F. Lanvin, R. Izquierdo, Runtime performance of `invokedynamic`: an evaluation with a Java library, IEEE Software 31 (4) (2014) 82–90.

[28] T. Würthinger, C. Wimmer, L. Stadler, Dynamic code evolution for Java, in: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ'10, ACM, New York, NY, USA, 2010, pp. 10–19.

[29] T. Würthinger, C. Wimmerb, L. Stadler, Unrestricted and safe dynamic code evolution for Java, Science of Computer Programming 78 (5) (2013) 481–498.

[30] S. Subramanian, M. Hicks, K. S. McKinley, Dynamic software updates: a VM-centric approach, in: Proceedings of the 30^{th} ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, ACM, New York, NY, USA, 2009, pp. 1–12.

[31] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, J. Lü, Low-disruptive dynamic updating of Java applications, Information and Software Technology 56 (9) (2014) 1086–1098.

[32] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: B. Magnusson (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP'02, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 205–230.

[33] J. Andersson, T. Ritzau, Dynamic code update in JDRUMS, in: Proceedings of the ICSE00 Workshop on Software Engineering for Wearable and Pervasive Computing, 2000, pp. 1–3.

[34] J. Andersson, A deployment system for pervasive computing, in: Proceedings of the International Conference on Software Maintenance, SM, 2000, pp. 262–270.

[35] S. Malabarba, R. Pandey, J. Gragg, E. Barr, J. Fritz Barnes, Runtime support for type-safe dynamic Java classes, in: E. Bertino (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP'00, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 337–361.

[36] M. Pukall, C. Kästner, G. Saake, Towards unanticipated runtime adaptation of Java applications, in: Proceedings of the Asia-Pacific Software Engineering Conference, APSEC'08, 2008, pp. 85–92.

[37] A. Orso, A. Rao, M. J. Harrold, A technique for dynamic updating of Java software, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), Montreal, Canada, 2002, pp. 649–658.

[38] L. Pina, M. Hicks, Rubah: Efficient, general-purpose dynamic software updating for Java, in: 5^{th} Workshop on Hot Topics in Software Upgrades, USENIX, San Jose, CA, 2013, pp. 1–6.

[39] ZeroTurnAround, JRebel, reload code changes instantly, https://zeroturnaround.com/software/jrebel (2018).

[40] J. Kabanov, Reloading Java Classes 401: HotSwap and JRebel Behind the Scenes, https://zeroturnaround.com/rebellabs/reloading_java_classes_401_hotswap_jrebel (2018).

[41] A. R. Gregersen, B. N. Jørgensen, Dynamic update of Java applications - Balancing change flexibility vs programming transparency, Journal of Software Maintenance and Evolution 21 (2) (2009) 81–112.

[42] A. R. Gregersen, D. Simon, B. N. Jørgensen, Towards a Dynamic-update-enabled JVM, in: Proceedings of the Workshop on AOP and Meta-Data for Software Evolution, RAM-SE '09, ACM, New York, NY, USA, 2009, pp. 2:1—-2:7.

[43] G. Kiczales, J. des Rivires, D. G. Bobrow, The Art of the Metaobject Protocol, The MIT Press, Cambridge, 1991.

[44] W. Taha, T. Sheard, Multi-stage programming with explicit annotations, in: Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '97, ACM, New York, NY, USA, 1997, pp. 203–217.

[45] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini, Fickle: Dynamic object re-classification, in: J. L. Knudsen (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP'01, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 130–149.

[46] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca, A type preserving translation of Flickle into Java, Electronic Notes in Theoretical Computer Science 62 (2002) 69–82.

[47] M. Serrano, Wide classes, in: R. Guerraoui (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP'99, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 391–415.

[48] M. Serrano, Bigloo. A practical Scheme compiler. User manual for version 4.2b., http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.pdf (2018).

[49] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca, A provenly correct translation of Fickle into Java, ACM Transactions on Programming Languages and Systems 29 (2007) 1–67.

[50] C. Chambers, Predicate classes, in: O. M. Nierstrasz (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP'93, Springer, Berlin, Heidelberg, 1993, pp. 268–296.

[51] C. Chambers, Object-oriented multi-methods in Cecil, in: O. L. Madsen (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP'92, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 33–56.

[52] G. Hjálmtýsson, R. Gray, Dynamic C++ classes: A lightweight mechanism to update code in a running program, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '98, USENIX Association, Berkeley, CA, USA, 1998, pp. 1–6.

[53] J. M. Redondo, F. Ortin, J. M. C. Lovelle, Optimizing reflective primitives of dynamic languages, International Journal of Software Engineering and Knowledge Engineering 18 (6) (2008) 759–783.

[54] J. M. Redondo, F. Ortin, Efficient support of dynamic inheritance for class- and prototype-based languages, Journal of Systems and Software 86 (2) (2013) 278–301.

[55] F. Ortin, J. M. Redondo, I. Lagartos, Efficient runtime metaprograming services for Java (support material website), `http://www.reflection.uniovi.es/invokedynamic/download/2018/jss` (2018).

[56] Oracle, Java 8 functional interfaces, `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html` (2018).

[57] I. Lagartos, JMPLib documentation, `https://computationalreflection.github.io/JMPLib` (2018).

[58] D. van Bruggen, Java Parser and Abstract Syntax Tree for Java 9, `https://github.com/javaparser/javaparser` (2018).

[59] G. Erich, H. Richard, J. Ralph, V. John, Design patterns: elements of reusable object-oriented software, Addison-Wesley Professional Computing Series, 1995.

[60] Oracle, Java 8 ToolProvider, `https://docs.oracle.com/javase/7/docs/api/javax/tools/ToolProvider.html` (2018).

[61] OW2, ASM Java bytecode manipulation and analysis framework, `http://asm.ow2.org` (2018).

[62] N. Nystrom, M. Clarkson, A. Myers, Polyglot: An Extensible Compiler Framework for Java, in: G. Hedin (Ed.), Compiler Construction, Vol. 2622 of Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2003, pp. 138–152.

[63] J. F. Roddick, A survey of schema versioning issues for database systems, Information and Software Technology 37 (7) (1995) 383–393.

[64] J. Kleinöder, M. Golm, MetaJava - a platform for adaptable operating-system mechanisms, in: Proceedings of the Workshops on Object-Oriented Technology, ECOOP '97, Springer-Verlag, London, UK, UK, 1998, pp. 507–514.

[65] The Jython project, Jython: Python for the Java platform, http://www.jython.org (2018).

[66] Mozilla, Rhino, an open-source implementation of JavaScript written in Java, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino (2018).

[67] Apache Groovy, Groovy, a multi-faceted language for the Java platform, http://groovy-lang.org (2018).

[68] J. Quiroga, F. Ortin, SSA transformations to facilitate type inference in dynamically typed code, The Computer Journal 60 (9) (2017) 1300–1315.

[69] M. Garcia, F. Ortin, J. Quiroga, Design and implementation of an efficient hybrid dynamic and static typing language, Software: Practice and Experience 46 (2) (2015) 199–226.

[70] A. Rigo, M. Fijalkowski, C. F. Bolz, A. Cuni, B. Peterson, A. Gaynor, H. Ardo, H. Krekel, S. Pedroni, PyPy official homepage, http://pypy.org (2018).

[71] J. M. Redondo, F. Ortin, A comprehensive evaluation of widespread Python implementations, IEEE Software 34 (4) (2015) 76–84.

[72] Google Inc., The V8 JavaScript engine, https://code.google.com/p/v8 (2018).

[73] Mozilla, The SpiderMonkey JavaScript engine, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey (2018).

[74] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: B. Magnusson (Ed.), European Conference on Object-Oriented Programming (ECOOP), Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 205–230.

[75] M. Golm, Design and implementation of a meta architecture for Java, Ph.D. thesis, Institute for Mathematical Machines and Data Processing of the Friedrich Alexander Universityl (January 1999).

[76] R. Pozo, B. Miller, Scimark 2.0: How fast is your Java platform for number crunching?, http://math.nist.gov/scimark2 (2018).

[77] I. Gouy, Shootout, the computer language benchmarks game, http://benchmarksgame.alioth.debian.org (2018).

[78] EPCC, Java Grande benchmark suite, https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite (2018).

[79] B. Leijdekkers, Metrics reloaded: automated code metrics plugin for IntelliJ IDEA, https://github.com/BasLeijdekkers/MetricsReloaded (2018).

[80] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: Proceedings of the 22^{nd} Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA, ACM, New York, NY, USA, 2007, pp. 57–76.

[81] D. J. Lilja, Measuring computer performance: a practitioner's guide, Cambridge University Press, 2005.

[82] MicrosoftTechnet, Windows server techcenter: Windows performance monitor, http://technet.microsoft.com/en-us/library/cc749249.aspx (2015).

[83] Microsoft, Windows management instrumentation, http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx (2015).

[84] R. H. R. Pereira, J. B. G. Perez-Schofield, F. Ortin, Modularizing application and database evolution – an aspect-oriented framework for orthogonal persistence, Software: Practice and Experience 47 (2) (2017) 193–221.