Computer Science and Information Systems 11(1):47–68 DOI: 10.2298/CSIS130129018C

JINDY: a Java library to support `invokedynamic`

Patricia Conde and Francisco Ortin

University of Oviedo, Computer Science Department
Calvo Sotelo s/n, 33007, Oviedo, Spain
{condepatricia,ortin}@uniovi.es

Abstract. Java 7 has included the new `invokedynamic` opcode in the Java virtual machine. This new instruction allows the user to define method linkage at runtime. Once the link is established, the virtual machine performs its common optimizations, providing better runtime performance than reflection. However, this feature has not been offered at the abstraction level of the Java programming language. Since the functionality of the new opcode is not provided as a library, the existing languages in the Java platform can only use it at the assembly level. For this reason, we have developed the JINDY library that offers `invokedynamic` to any programming language in the Java platform. JINDY supports three modes of use, establishing a trade-off between runtime performance and flexibility. A runtime performance and memory consumption evaluation is presented. We analyze the efficiency of JINDY compared to reflection, the `MethodHandle` class in Java 7 and the Dynalink library. The memory and performance costs compared to the `invokedynamic` opcode are also measured.

Keywords: `invokedynamic`, Java Virtual Machine, dynamically generated classes, reflection, runtime performance.

1. Introduction

The Java platform offers a set of features, such as platform independence and adaptive HotSpot JIT compilation, which has led to its widespread use and general acceptance. These features have contributed to the proliferation not only of libraries, frameworks or applications, but also of languages implemented on the Java platform. An existing research has identified more than 240 language implementations on this platform [28], including some well-known examples such as Scala, JRuby, Groovy, Jython and Clojure.

Since its first release, the Java platform has gradually incorporated more features commonly supported by dynamic languages. 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 dynamically funneling all the method calls of that interface to an `InvocationHandler`. The Java `instrument` package (Java 5) provides services that allow 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. The last version, 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 [21].

Until the inclusion of the `invokedynamic` opcode, the dynamically typed languages implemented on the Java platform had to simulate the dynamic features that the platform did not support. Reflection was the main feature to obtain dynamic method invocation. However, this simulation usually causes a runtime penalty [24]. The key difference between `invokedynamic` and the reflection API is the runtime checks upon method access. Reflection performs these checks at every method call, whereas `invokedynamic` perform this check upon method handle creation [28]. Another difference is the possibility to avoid type conversions performed by the reflective approach (i.e., avoiding the use of `Object`). These differences suggest that `invokedynamic` can be used to improve the runtime performance of dynamic language implementations [33] and any library, framework or Java application that makes significant use of reflection.

Despite the expected benefits of the new `invokedynamic` opcode, the Java programming language has not been changed to support this feature. Furthermore, the new `java.lang.invoke` package in Java 7 [22] does not provide a high-level mechanism to use `invokedynamic` from within the platform. The new bytecode is expected to be used mainly by compilers that generate binary code including the new JVM instruction. However, this fact makes it difficult to access this functionality from any high-level language on the platform. A first use case from a high-level language could be as an alternative mechanism to reflection. There could be some other scenarios, such as the dynamic separation of aspects [23] and the implementation of multi-methods [6].

The main contribution of this paper is the development of a Java library, called JINDY, to support the `invokedynamic` opcode from any high-level language on the Java platform. Additionally, we assess its runtime performance, compared to the existing approaches. We also measure the penalty introduced by our library in comparison with the direct use of the `invokedynamic` opcode.

This paper is structured as follows. Section 2 describes the `invokedynamic` opcode and the dynamic code generation technique used in the development of JINDY. The library is described in Section 3, and Section 4 evaluates its runtime performance and memory consumption. Section 5 discusses the related work and Section 6 concludes and presents the future work.

2. Context

2.1. The `invokedynamic` instruction

The `invokedynamic` JVM opcode provides a new user-defined dynamic linkage mechanism, postponing type checking of method invocation until runtime. As shown in Figure 1, an `invokedynamic` instruction is initially in an *unlinked state* (state 0 in Figure 1). That means that each `invokedynamic` instruction (`main` method of the `Application` class in Figure 1) does not statically specify the method responsible for conducting the dynamic method selection. This method, in the `invokedynamic` documentation, is called *bootstrap* method (`method` of the `Bootstrap` class in Figure 1).

An `invokedynamic` instruction is *linked* just before its first execution (state 1 in Figure 1). At runtime, the JVM calls the bootstrap method, evaluating the names of the class and method (passed as strings). The bootstrap method is responsible for returning the method dynamically resolved. The dynamic resolution consists in returning a `CallSite`

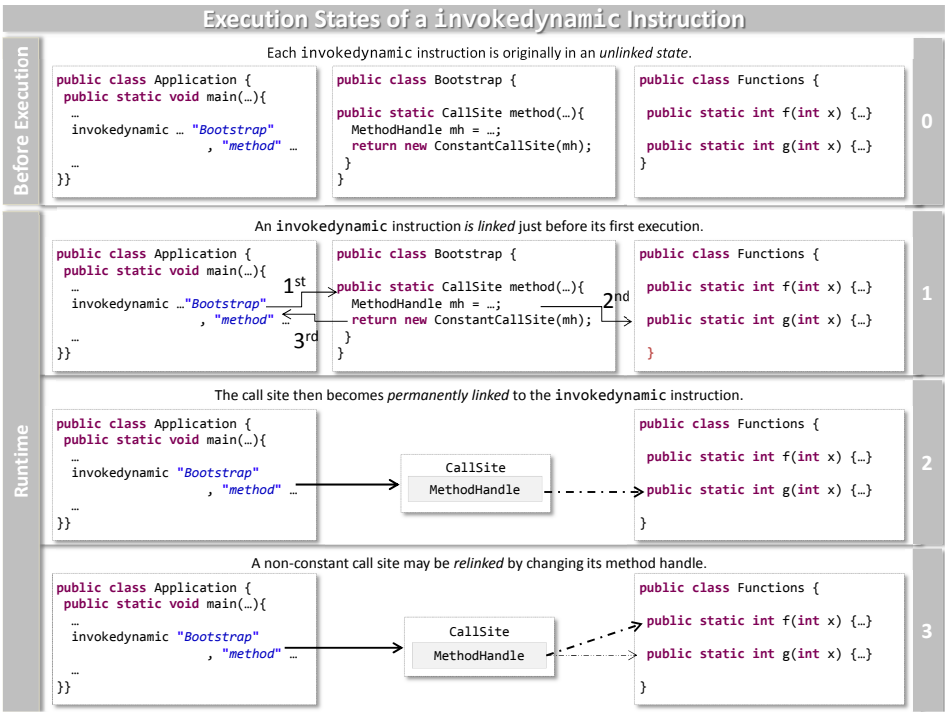


Fig. 1. Runtime execution of `invokedynamic`.

holding the `MethodHandle` that represents the selected method (`g` of the `Functions` class in Figure 1).

An `invokedynamic` instruction is considered *permanently linked* when the bootstrap method returns a `CallSite` (state 2 in Figure 1). From that time on, the following invocations simply call the method referenced by the returned `MethodHandle`. Besides, the JVM will apply the usual optimizations performed for common statically typed method invocations.

It could be necessary to *relink* the linked method, due to some event occurred at runtime (state 3 in Figure 1). After relinking, the `MethodHandle` will point to `Functions.f` instead of to the original `Functions.g` method. For this purpose, the new `invoke` package [22] offers three implementations of the `CallSite` abstract class: the `ConstantCallSite` class for permanent method handles, and the `Volatile-CallSite` and `MutableCallSite` classes for relinking a method handle with the same type descriptor (`MethodType`) –`VolatileCallSite` supports multi-threading.

As mentioned, a resolved `invokedynamic` instruction is linked to a `CallSite`, and each `CallSite` contains a `MethodHandle`. This `MethodHandle` is a reference to an underlying method, field or constructor. Therefore, an `invokedynamic` instruction performs the appropriate operation associated to the corresponding `Method-Handle`.

2.2. Dynamic code generation

Since `invokedynamic` is a JVM opcode, JINDY generates JVM code to provide its functionality. Moreover, the code generation process should be performed dynamically, because this opcode postpones linking until runtime.

ASM[1] [5] is a code generation framework used to implement dynamic languages such as JRuby [20], Jython [12] and Groovy [15]. It has also been used to implement other code transformation tools such as JooFlux [26] and Soot [4]. This framework can be used to modify existing classes and dynamically generate new ones, managing their binary representation. ASM 4.0 supports the new `invokedynamic` opcode. For all these reasons, we selected ASM to dynamically generate JVM code in the implementation of JINDY.

Figure 2 shows the process we follow to support `invokedynamic` from any high-level language on the Java platform. An application calls JINDY (1) that, in turn, uses ASM to generate JVM code at runtime (2). The generated code includes a class that implements the `Callable<T>` interface shown in Figure 2. The generated `invoke` method consists in an `invokedynamic` JVM instruction specifying a bootstrap method. The parameters JINDY passes to ASM to generate the appropriate `invokedynamic` statement are: a `String` with the symbolic name of the method to be invoked; another `String` with its type descriptor; and a `Handle` to identify the bootstrap method (the names of the bootstrap class and method). More parameters can optionally be passed.

Once the binary class is generated, it is dynamically loaded and instantiated (3). To load the binary class, JINDY implements a `ClassLoader` that loads the class after generating it. Once loaded, our library instantiates the class by calling its constructor (using reflection), returning a `Callable<T>` object (4). When the programmer calls the `invoke` method of this interface (5), the `invokedynamic` opcode will be finally executed by the JVM (as described in Figure 1), since this opcode is the generated method body. The first parameter of `invoke` is the object the underlying method is invoked from, i.e. `this` (`null` if the method is `static`), and the rest of optional parameters are the method arguments. As mentioned, an `invokedynamic` execution of an unlinked method causes the execution of the bootstrap method (6), its linkage (7) and the final execution.

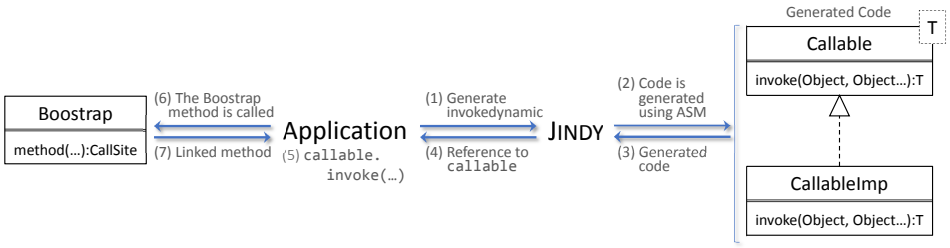


Fig. 2. The dynamic code generation process implemented by JINDY.

The dynamic code generation mechanism shown in Figure 2 has been previously applied in Groovy to perform dynamic method invocations [15]. Dynamic code generation

[1] http://asm.ow2.org

involves a runtime performance and memory consumption penalty, as discussed in Section 4.4. We have reduced this penalty by implementing a cache of instantiated classes, avoiding the generation existing classes.

3. JINDY

The implemented library allows the programmer to use the `invokedynamic` opcode from any language implemented on the Java platform. JINDY can be used in all the scenarios where `invokedynamic` may be useful. Programmers may create their own bootstrap methods to define how call sites must be dynamically linked. Besides, it also provides specific routines as a more efficient alternative to the reflection API. This section shows examples of using JINDY in these scenarios.

The motivating example uses JINDY to implement a dynamically adaptable dependency injection system. Dependency injection is a software design pattern that allows removing hard-coded dependencies between software components (*beans*) [10]. Dependencies are described in a separate document that is processed by the *injector* (sometimes called container, assembler or provider), which creates instances of the appropriate classes and interconnects them. We show how to perform this instantiation and interconnection with JINDY. Besides, we extend the injector to support the dynamic modification of object dependencies without changing the source code of the application, using `invoke-dynamic`.

3.1. JINDY as an alternative to the reflection API

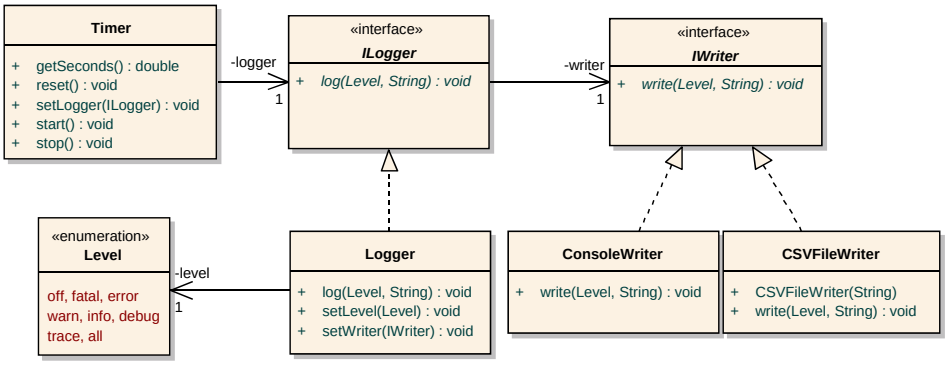
We first use JINDY to implement a typical dependency injection system (in the following subsections we extend it to make it dynamically adaptable). This first scenario shows how our library provides a more efficient alternative to the reflection API (see the performance evaluation in Section 4). Both component instantiation and interconnection are done using JINDY.

The top of Figure 3 shows an example class diagram, where a `Timer` class provides a set of methods. When these methods are called, the timer logs information by using its associated `logger`. Similarly, an `ILogger` invokes the `write` method of the associated `IWritter` (when the `level` of the message to be logged is greater or equal to the `level` field of the logger [1]). Figure 3 presents two types of `IWriters`: `ConsoleWriter` to show messages on the console, and `CSVFileWrite` to write information in comma-separated value text files.

We have used an XML document format similar to the Spring Framework to declare the dependencies between components [29]. The example document in the bottom of Figure 3 shows how a `ConsoleWriter` instance is assigned to the `writer` field of the `Logger` class, and a `Logger` instance is also associated to a `Timer`. The dependency injector will instantiate these objects, associating them as stated in the XML document. Therefore, the object dependencies are not hard-coded in the application source code.

Figure 4 shows an excerpt of our first dependency injection system (the full source code is available for download[2]). The main `Application` class (bottom left corner)

[2] `http://www.reflection.uniovi.es/invokedynamic/Jindy`




```
<beans xmlns="http://www.springframework.org/schema/beans"
 … >
    <bean id="writer" class="ConsoleWriter" />
    <bean id="logger" class="Logger"><property name="writer" ref="writer" class="IWriter"/></bean>
    <bean id="timer" class="Timer">  <property name="logger" ref="logger" class="ILogger"/></bean>
</beans>
```

Fig. 3. Example application and its dependency injection XML document.

asks the `BeanInjector` to process the XML document in order to create the class instances and inject their dependencies. Afterwards, the `timer` bean is retrieved and used. The `BeanInjector` (top left of Figure 4) contains an `ObjectTable` collecting all the bean instances. Upon construction, the `BeanInjector` calls the `addBeanId` method of this `ObjectTable` (bottom right) for each `bean` element in the XML document. The `addBeanId` method shows the first example use of JINDY. The `ProxyFactory` class is provided to use the `invokedynamic` opcode through a set of methods. The bean class is first loaded, and the `generateConstructor` method dynamically generates a method that invokes the default constructor[3] using `invokedynamic`; additionally, the corresponding bootstrap method is also generated by JINDY.

The `BeanInjector` also interconnects class instances with the `inject` method (Figure 4, top left). This is a second example of using JINDY for dynamic method invocation. Once the two associated objects and their classes are retrieved, our library is used to call the appropriate *setter* method (setter injection [10]). The `generateCallable` method generates a new class at runtime, implementing the generic `Callable<T>` interface provided by JINDY (Figure 4, top right). The generated class overrides the `invoke` method, calling the specified setter method (as described in Figure 2). The generic `T` type of the `Callable` interface corresponds to the return type of the `invoke` method. Generics are used to avoid casting the return value, on contrast to reflection.

These two examples shows how JINDY can be used as an alternative to the reflection API. The main difference between JINDY and reflection is that JINDY uses `invokedynamic` by means of dynamic code generation, whereas reflection uses introspection. Besides, JINDY implements a cache of the instantiated classes, performing better than reflection (Section 4).

[3] In our dependency injection example system, bean classes must support a constructor without parameters.

Fig. 4. Dynamic method invocation using JINDY.

3.2. Avoiding type conversions

In the previous section, the `Callable` JINDY interface receives `Object` type arguments, similar to the reflection methods. This produces many runtime type conversions that involve a runtime performance penalty. However, there are scenarios in which the parameter and return types can be known at compile time. For these cases, JINDY offers a set of services to avoid type conversions, improving code robustness (static type checking) and runtime performance (Section 4).

In the previous version of the dependency injection system, instances were created and interconnected at startup, but changing the XML document at runtime did not imply changing the running application. We now extend the example to allow the dynamic modification of the application, permitting the user to instantiate new beans and changing their interconnections at runtime without changing the application source code. Therefore, `invokedynamic` is used to *relink* the method associated to specific `CallSites`. In particular, we modify the meaning of field accesses (i.e., `this.logger` and `this.writer`) changing the returned objects depending on the contents of the XML dependency injection document.

Figure 5 shows the new `Logger` class[4]. JINDY returns a `GetWriter` object that implements the access to the `writer` field (i.e., `this.writer`) as a method. The developer defines the `GetWriter` interface declaring a method (`invoke`) with the same signature as the getter method of the `writer` field. JINDY dynamically generates a class implementing `GetWriter`, which uses `invokedynamic` to access the `writer` field.

[4] The `bootstrap` object will be discussed in the following subsection.

Since no type conversion is performed at runtime, our library performs significantly better than reflection (see Section 4).

```
public class Logger implements ILogger  {

  private IWriter writer;
  private Level level = Level.ALL;

  public void log(Level level, Object message) {
    if (this.level.ordinal() >= level.ordinal())
      ProxyFactory.<GetWriter>generateInvokeDynamic(bootstrap, GetWriter.class).invoke(this)
                  .write(level, message);
  }

  private static Bootstrap bootstrap = new Bootstrap("Bootstrap", "bootstrapMethod",
                                                     Logger.class, "writer");
}
```

```
public interface GetWriter {
  IWriter invoke(Logger logger);
}
```

Fig. 5. Avoiding type conversions in JINDY.

Notice how the `Logger` class provides no setter method. With this new approach of dependency injection, neither setter methods no parameterized constructors need to be implemented (i.e., setter and construction injection). The bean injector modifies the semantics of field access, injecting the dependencies transparently depending on the XML document, and considering its runtime changes.

3.3. Providing a custom bootstrap method

As shown in Figure 5, the programmer provides a bootstrap method to define how method (or field access) linkage must be performed. Figure 6 details the implementation of the bootstrap method. First, the XML document is read and loaded into the object table (Section 3.1). Second, a getter method implementation is dynamically generated. This getter method will be used to define the new semantics of field access, considering the dependencies declared in the XML document. Third, a `VolatileCallSite` wrapping this method is created; it is volatile because its wrapped method will be asynchronously modified at runtime. Finally, the call site is stored in a `callSiteTable` to allow the subsequent access, and returned.

In Figure 6, the new `Application` runs the bean injector as a daemon thread, while the application keeps running. The `BeanInjector` thread (i.e., its `run` method) analyzes the XML document each 500 milliseconds (`processXML`), only if its modification date has been changed. If one of the dependencies is modified, the `setBeanProperty` method will be called by `processXML`. `setBeanProperty` gets the `VolatileCallSite` stored by the bootstrap, generates the corresponding getter method implementation (called `getField`), and dynamically changes the target method of the call site to the one generated (`getField`). The result is an asynchronous modification of the associated object (dynamically adaptable dependency injection), without changing the application source code. Another benefit is that new implementations of `IWriter` and

ILogger can be added and injected at runtime, without stopping the application execution. For instance, the example source code provided[5] adds a new HtmlTextWriter implementation of IWriter at runtime, and injects one instance as the new writer property of the logger bean.

```
public class Bootstrap {
  public static CallSite bootstrapMethod(Lookup lookup, String name,
                              MethodType methodType, Class<?> klass, String member) {
  beanInjector.processXML();
  MethodHandle mh = beanInjector.generateGetter(klass, member, methodType);
  VolatileCallSite callsite = new VolatileCallSite(mh);
  beanInjector.callSiteTable.put(new FieldKey(klass, member, methodType.returnType()),
                                 callsite);
  return callsite;
  }
}
```

```
public class Application {
  public static void main(String... args) {
    BeanInjector injector =
            new BeanInjector("beans.xml");
    Timer timer = injector.<Timer>getBean(
                                  "timer");
    injector.start();

    while(true) {
      timer.start(); Thread.sleep(1000);
      timer.stop();  timer.getSeconds();
      timer.reset();
} } }
```

```
public class BeanInjector extends Thread {

  public void run() {
    while(true) {
      Thread.sleep(500); this.processXML();
    }
  }

  Map<FieldKey, VolatileCallSite> callSiteTable =
        new HashMap<FieldKey, VolatileCallSite>();

  public void setBeanProperty(Class<?> klass, String fieldName,  Class<?> fieldClass,
                              String concreteFieldType) {
    FieldKey fieldKey = new FieldKey(klass, fieldName, fieldClass);
    VolatileCallSite callsite = callSiteTable.get(fieldKey);
    String generatedClassName = this.generateGetter(klass, fieldName, fieldClass, concreteFieldType);
    MethodHandle mh = MethodHandles.lookup().findStatic(Class.forName(generatedClassName),
                                          "getField", MethodType.methodType(fieldClass, klass));
    callsite.setTarget(mh);
  }
  …
}
```

Fig. 6. Using JINDY with a custom bootstrap method.

We have seen how the JINDY library provides high-level support for invokedynamic. The library allows postponing until runtime class and instance method invocations, constructor calls, and instance and static field accesses. It also supports user-defined bootstrap methods to allow customized linkage. We have also shown how JINDY can be used to dynamically relink methods, permitting the asynchronous adaptation of call sites. A more detailed documentation of the library can be consulted in [7].

4. Evaluation

We have evaluated JINDY in the three use cases described in Section 3: as an alternative to reflection, avoiding type conversions, and providing custom bootstrap methods (using constant call sites). We measure the runtime performance and memory consumption of

[5] http://www.reflection.uniovi.es/invokedynamic/Jindy

JINDY compared to reflection, `MethodHandle` and the Dynalink library [32]. Additionally, we have measured the performance penalty of JINDY compared to the direct use of `invokedynamic`, as well as the comparison of this new opcode to the corresponding statically typed JVM instructions.

4.1. Methodology

A synthetic micro-benchmark has been developed to evaluate the above mentioned use cases. This micro-benchmark takes into account the following scenarios:

1. The `invokedynamic` opcode. We compare the performance of the `invokedynamic` opcode with JINDY, its corresponding statically typed opcode, reflection, the `invokeExact` method of `MethodHandle`, and Dynalink. Dynalink is Java library that allows performing dynamic operations on objects without knowing their static types [32]. It uses the services of the new Java 7 `java.lang.invoke` package, including `invokedynamic` (a more detailed description is presented in Section 5).
2. JINDY, as an alternative to the reflection API. In this case, we compare different types of reflective scenarios. We measure the execution time used to obtain classes and methods using introspection, apart from its invocation. JINDY is also used to optimize a real third-party application, measuring the performance improvement obtained.
3. Performance and memory cost. We measure the increase of the memory consumption and runtime performance of JINDY, compared to the direct use of the `invokedynamic` opcode.

The developed synthetic micro-benchmark evaluates instance, class and interface method invocations (one single parameter), as well as instance and class field accesses (get and set). The methods simply increment a field value (receiving another value as a parameter) and return the value of the incremented field. The return values of methods and fields are assigned to local variables.

These scenarios have been implemented in JINDY following the three different use cases described in Section 3. Firstly, as an alternative to the reflection API, performing type conversions (JINDY_TC in Figures 7, 8, 9 and 10). The second use case utilizes customized interfaces to avoid type conversions, without providing a bootstrap (JINDY_I). The last scenario (JINDY_BI) provides a custom bootstrap method, equivalent to those used in the other two scenarios (i.e., returning constant call sites).

Regarding the data analysis, we have followed the methodology proposed in [11] 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 start-up JIT compilation does not involve a significant variability in the total running time, and Hotspot dynamic optimizations have been 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 have taken $p = 30$) 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 computed using the *Student's* t-distribution because we took $p = 30$ [17]. Therefore, we compute the confidence interval $[c_1, c_2]$ as:

$$c_1 = \overline{x} - t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}}$$

Where $\overline{x}$ is the arithmetic mean of the x_i measurements, $\alpha = 0.05(95\%)$, s is the standard deviation of the x_i measurements, and $t_{1-\alpha/2;p-1}$ is defined such that a random variable T, that follows the *Student's* t-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1 - \alpha/2$.
The data provided is the mean of the confidence interval plus a percentage indicating the width of the confidence interval relative to the mean.

The steady-state methodology comprises 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 this state when the coefficient of variation (CoV, defined as the standard deviation divided by the mean) of the last $k+1$ iterations (from $s_i - k$ to s_i) falls below a threshold (2%).
3. For each application execution, we compute the mean x_i of the $k+1$ benchmark iterations under steady state:

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

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 computer over the $\overline{x_i}$ measurements.

To measure execution time of each benchmark invocation (in the steady-state methodology) we have instrumented 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 operating system Performance and Reliability Monitor [37]. We measure the difference between the beginning and the end of each benchmark invocation to obtain the execution time of each benchmark run.

The memory consumption has been measured following the start-up methodology to determine the memory used by the whole process. For that purpose, we have used 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. These 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` has been measured with explicit calls to the services of the Windows Management Instrumentation infrastructure [18].

All the tests were carried out on a lightly loaded 2.40 GHz Intel Core i5 2430M system with 4 GB of RAM running an updated 64-bit version of Windows 7 Home Premium SP1. We have used the Java Standard Edition 1.7 update 9 for 64 bits.

4.2. Evaluation of the `invokedynamic` opcode

We have evaluated the performance of `invokedynamic`, JINDY in the 3 use cases defined (JINDY_I, JINDY_TC and JINDY_BI), the corresponding statically typed opcodes, reflection, `invokeExact` of `MethodHandle` and Dynalink. The 7 operations of the synthetic micro-benchmark (instance, class and interface method; and class and instance get and set field) have been executed in loops from 1 to 10,000 million iterations.

Since the test that uses `invokedynamic` cannot be directly written in Java, the following method has been followed to implement it. We first take the statically typed approach that measures the 7 operations mentioned above. This program is then disassembled with the ASMifier tool of ASM, replacing the 7 statically typed opcodes (`invoke{virtual, interface, static}` and `{get, set}{field, static}`) with the appropriate `invokedynamic` instruction. We implement 7 bootstrap methods for the 7 types of `invokedynamic` operations. Finally, ASM is used to convert the new assembly code into the JVM `.class` files.

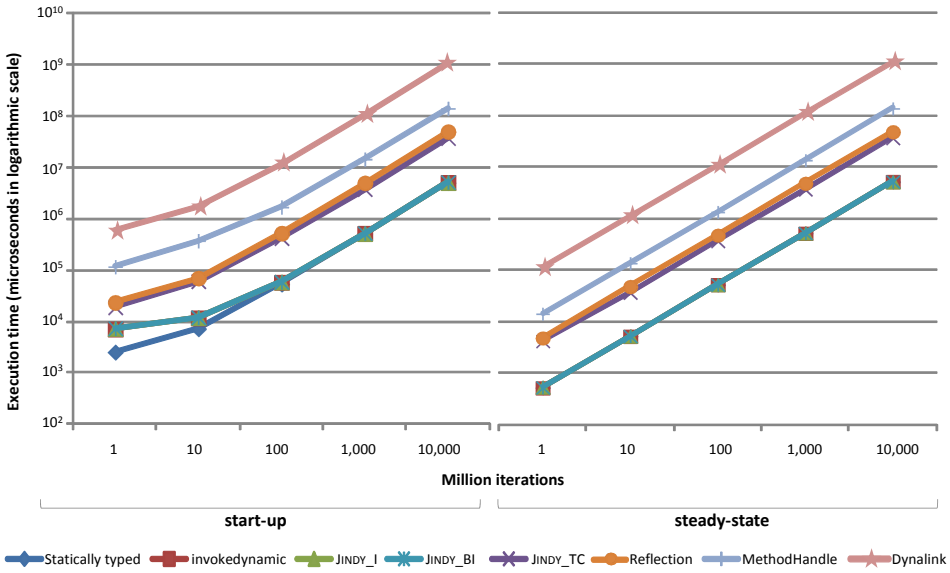


Fig. 7. Average execution time of the primitive operations (microseconds in logarithmic scale).

Figure 7 shows the average execution time obtained for the 7 operations. The results for short-running (start-up) applications indicate that, for 1 million iterations, the new invokedynamic opcode is 181.2% slower than the corresponding statically typed opcode. The penalty of using JINDY_I compared to invokedynamic is 2.49%, being 0.54% and 61.28% faster than JINDY_BI and JINDY_TC, respectively. Compared to reflection, MethodHandle and Dynalink, JINDY_I is 2.22, 15.11 and 83.36 times faster, respectively.

Figure 7, start-up, also shows how the performance of JINDY and invokedynamic improves as the number of iterations increases. When reaching 100 million iterations, the benefit of JINDY_I and invokedynamic regarding reflection, MethodHandle and Dynalink grows to 8.03, 27.88 and 197 factors, respectively. For the same number of iterations, statically typed opcodes performs 0.51% and 8.57% better than invoke-dynamic and JINDY_I, respectively.

For long-running applications (steady-state in Figure 7), runtime performance shows lower dependence on the number of iterations. For 1 million iterations, the smallest difference between JINDY_I, JINDY_BI, invokedynamic and the statically typed invocations is lower than 2.5%. As the number of iterations increases, the differences lightly decreases, reaching values lower than 0.5% for 10,000 million iterations. In the start-up methodology, the JVM reaches its steady state with 1,000 million iterations. At this state, the results for short-running applications are very similar to those obtained for server applications (Figure 7).

Figure 8 shows the execution times of the seven operations for 1,000 million iterations, running long-running applications (steady-state). Performance values have been divided by those obtained by the corresponding statically typed opcode. For all the operations, differences among invocations using the statically typed opcode, invoke-dynamic, JINDY_I, and JINDY_BI are barely appreciable. Average penalties of JINDY_I and JINDY_BI compared to invokedynamic are 0.48% and 0.79%, with a standard deviation of 1.45% and 0.68%, respectively.

When using reflection, the worst results are obtained getting and setting field values. For example, setting the value of instance fields using reflection is 13.72 and 1.97 times slower than JINDY_I and JINDY_TC, respectively. A similar behavior is seen with invokeExact of MethodHandle, where reading an instance field is 50.58 times slower than the static opcode. The smallest difference between JINDY and the rest of approaches is exhibited in the invocation of static methods. In this case, JINDY_TC and reflection obtain practically the same performance, and JINDY_TC is 370.53% and 1,763% faster than MethodHandle and Dynalink. In this scenario, JINDY_I and JINDY_BI are 824.20% and 821.34% faster than reflection; perform 25 and 24.92 times better than MethodHandle; and are 168 factors faster than Dynanlink.

The results obtained with JINDY_TC for method invocation confirm the cost of type of conversions (the argument and return types are long). The penalties of JINDY_TC compared to reflection are 3.53%, 3.67% and 1.42% for instance, class and interface methods invocation, respectively. However, if type conversions are avoided (JINDY_I), our library turns out to be 713.07%, 713.76% and 797.38% faster.

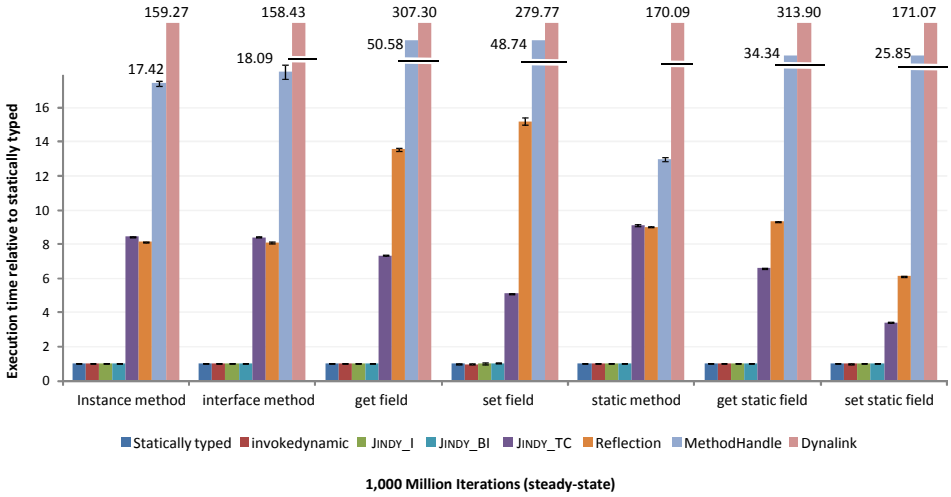


Fig. 8. Execution time per primitive operation relative to statically typed opcodes.

4.3. JINDY evaluation as an alternative to the reflection API

In the previous subsection, reflective method invocations have been measured considering the execution time of the `invoke` method in the `Method` class. However, the execution time of obtaining the class and the appropriate method using reflection have not been measured. Similarly, we have not evaluated the execution time of the dynamic class generation performed by JINDY. Therefore, in this section we consider the cost of these operations. Prior to the `invokedynamic` execution, we obtain an instance of the class that supports this invocation, using the JINDY `ProxyFactory` factory methods defined in Figure 4. These operations are now included in the micro-benchmark, for the three use cases of JINDY (JINDY_I, JINDY_TC and JINDY_BI) and reflection. The `MethodHandle` and Dynalink approaches are not considered here because of their low runtime performance.

Figure 9 shows the average execution time obtained for the 7 operations. Iterations have been reduced to 1,000 million due to the high execution times obtained. Execution times are computed as the arithmetic mean of the 7 operations described in Section 4.1. As in the previous subsection, runtime performance of short- (start-up) and long-running (steady-state) applications show a different dependence on the number of iterations. Long-running applications do not show a significant difference when increasing the number of iterations. For 1 million iterations JINDY_I, JINDY_TC and JINDY_BI are 23.02, 16.55 and 6.29 times faster than reflection, respectively. JINDY_BI is the scenario where the lowest benefit is obtained, because of the penalty caused by the use of custom bootstrap methods.

For short-running applications (start-up), the performance benefit of JINDY compared to reflection increases as the number of iterations grows. For 1 million iterations, JINDY_I, JINDY_TC and JINDY_BI obtain a performance benefit of 340.07%, 283.06% and 231.66%, respectively, compared to reflection. With 1,000 million iterations these benefits grow to 1,197%, 973% and 565%.

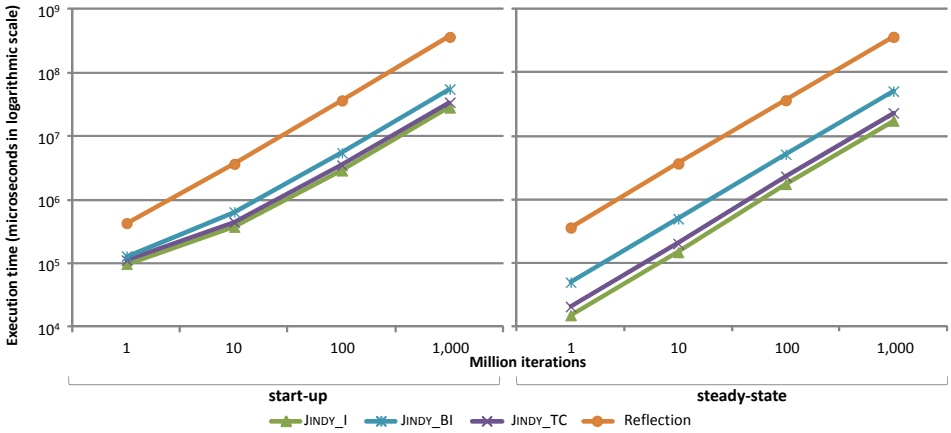


Fig. 9. Average execution time for reflective scenarios (microseconds in logarithmic scale).

Figure 10 shows the results per operation for 1,000 million iterations, when the application has reached a steady state. The results have been divided by the execution time of JINDY_BI. The major benefits of using JINDY, as an alternative to reflection, are obtained when getting field values of both instances (JINDY_I, JINDY_BI and JINDY_TC are 24.76, 7.75 and 21.75 times faster than reflection) and classes (27.05, 8.34 and 23.78 times faster than reflection). The lowest performance benefit is obtained when invoking interface methods, being JINDY_I, JINDY_BI and JINDY_TC 9.62, 2.67 and 6.01 times faster than reflection.

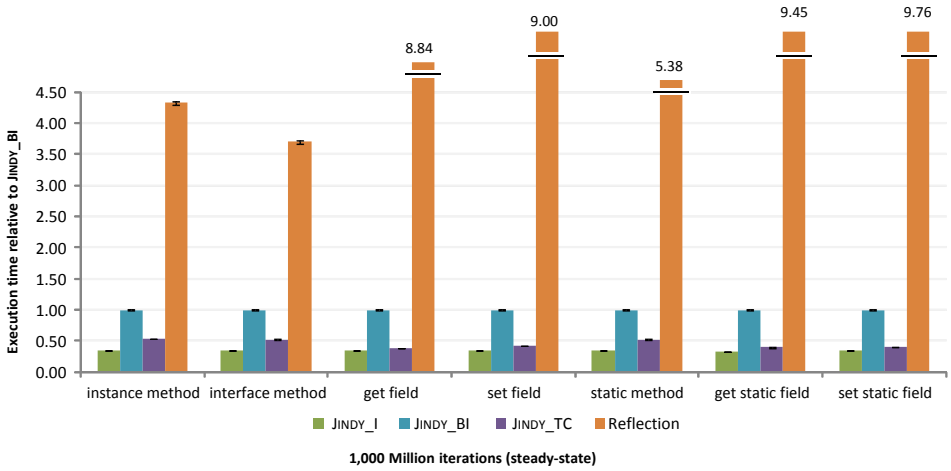


Fig. 10. Execution time per primitive operation relative to JINDY_BI, for reflective scenarios.

Comparing this data with the one presented in the previous subsection, we can see have the benefit of JINDY in this case is even higher. The performance benefits are, on

average, 807.23% greater than those in the previous subsection. Therefore, the cost of generating the `invokedynamic` opcode seems to be lower than the reflective retrieval of a method and its class.

Evaluation of a real application

We have also evaluated the performance improvement of JINDY when applied to a real application that makes extensive use of reflection. In the previous synthetic micro-benchmark, only the execution time of reflective operations were measured. In this case, a real application executes both reflective and statically typed code. Therefore, we assess how JINDY can optimize the overall application execution.

The application we have selected for this test is OVal, a pragmatic and extensible validation framework for any kind of Java objects (not only JavaBeans) [35]. OVal allows defining dynamic constraints on Java properties, methods and constructors. The constraints can be defined with Java annotations or XML files. Constraint validation is performed by OVal at runtime, using the reflection API.

Analyzing the source code of OVal, we identified 16 reflective method invocations. We modified the OVal implementation, replacing these reflective operations with the appropriate invocations to the JINDY library. Besides, we created a third version using Dynalink instead of JINDY. The three different versions were executed, and their execution times were measured following the methodology described in Section 4.1.

The entity model of the example application is shown in Figure 11. The application creates three instances of `Person`, two `Banks`, and one instance of `Company`, `Job` and `Marriage`. We defined 28 constraints on properties and methods, using Java annotations. Example constraints are *parents of one person can be neither descendants nor his or her spouse*, and *a person cannot have two different jobs in the same company*[6]. OVal is programmatically called to check if the set of constraints are fulfilled at a specific point of execution (i.e., *on demand* validation). This validation is performed an increasing number of times.

Table 1 presents the execution times and error percentages of the three approaches, from 1 to 1 million invocations in the steady-state methodology (start-up showed high percentage errors). For 1 and 10 invocations, the difference between reflection and JINDY is lower than the error percentage (they cannot be considered to differ). For 100 invocations JINDY is 7.14% faster than reflection, and this benefit grows to 75.07% for 1,000 invocations. The benefit of JINDY compared to reflection seems to converge towards 79.18% (the benefit difference between 100,000 and 1 million invocations is 0.9%).

Table 1 also shows the execution time of OVal, when reflective invocations have been replaced with Dynalink. JINDY performs 324% better than Dynalink for one single invocation. This benefit grows up to 296 factors for one million invocations. Therefore, the dynamic generation of `invokedynamic` opcodes (the JINDY approach) compared to the use of `MethodHandles` (using Dynalink from Java) involves a significant performance benefit. In the execution of this test, JINDY has required 15% more memory resources than reflection, but 29.92% less memory than Dynalink.

[6] The rest of constraints can be consulted in the code available for download.

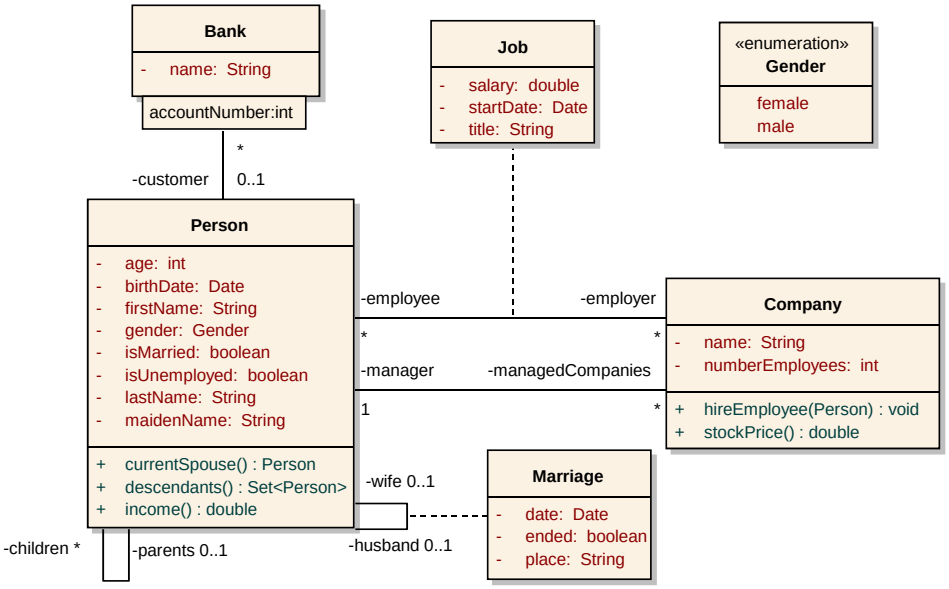


Fig. 11. Entity model used in the evaluation of the OVal framework.

4.4. Performance and memory cost

The dynamic generation of classes performed by JINDY (Section 2.2) produces a runtime performance and memory consumption penalty compared to the direct use of `invokedynamic`. To evaluate this penalty, we have measured the average execution time of the 7 common operations. In the computer used, the constant average execution time is 22,068.38 microseconds. However, as discussed in the previous subsections, the relative cost of JINDY compared to `invokedynamic` decreases as the number of invocations increases, becoming 2.59% for 1,000 invocations and 0.69% for 10 million invocations.

Using the developed micro-benchmark, we have also evaluated the memory consumption of our library. The average memory consumption of JINDY, when using custom interfaces (JINDY_I), is 2.55% higher than `invokedynamic`. Comparing it with the corre-

Invocations	JINDY	Reflection	Dynalink
1	8,342 ±5.2%	8,141 ±6.6%	35,358 ±6.6%
10	14,160 ±4.2%	13,653 ±3.8%	58,480 ±3.3%
100	25,207 ±5.9%	27,008 ±7.6%	314,659 ±1.9%
1,000	77,103 ±1.9%	134,985 ±1.9%	3,468,511 ±1.8%
10,000	876,993 ±1.6%	1,453,619 ±1.9%	147,744,785 ±2.1%
100,000	6,502,597 ±1.6%	11,592,832 ±2.4%	1,895,077,469 ±2.6%
1,000,000	63,674,864 ±1.7%	114,093,201 ±1.9%	18,950,774,685 ±3.6%

Table 1. Average execution time (microseconds) and error percentages of the OVal validation framework.

sponding statically typed opcode, reflection and `MethodHandle`, these increments are 8.31%, 7.47% and 5.94%, respectively; Dynalink requires 27.37% more memory than JINDY_I. Using our library in scenarios that require type conversions (JINDY_TC) and providing a custom bootstrap method (JINDY_BI), JINDY requires 0.057% and 1.30% more memory, respectively, than JINDY_I. These results were calculated with a 95% confidence level and error intervals lower than 2%.

5. Related Work

Dynalink is an `invokedynamic`-based high-level linking and meta-object protocol library [32]. Dynalink allows performing dynamic operations on objects without knowing their static types. It can be used from Java to facilitate the creation of `MethodHandles` that the programmer later invokes to perform the dynamic operations. It can also be used from a JVM assembler (e.g., ASM) to make the generation of `invokedynamic` instructions easier. Dynalink provides a set of bootstrap method implementations, and a set of conventions for common dynamic operations on objects. For instance, the `"dyn:getProp:color"` string represents the dynamic access to the `color` property (i.e., both the `color` field and the `getColor` method). Therefore, it provides a form of duck typing [27]. In its current version 0.6, Dynalink does not generate `invokedynamic` opcodes when used from Java; `MethodHandles` should be invoked instead. It provides services to facilitate the use of `invokedynamic` when used from a JVM assembler.

The Da Vinci Machine (or MLVM, Multi-Language Virtual Machine) [31] is an OpenJDK project aimed at extending the JVM with first-class architectural support for languages other than Java, especially dynamic ones. This project prototypes a number of extensions to the JVM, so that it can run non-Java languages efficiently, with a performance level comparable to that of Java itself. The Da Vinci project has several subprojects such as the JSR 292, aimed at supporting dynamically typed languages on the Java platform [30], including the new `invokedynamic` opcode. In 2010, Chanwit Kaewkasi conducted a preliminary study of the efficiency of `invokedynamic` in MLVM. He presented the results of running an `invokedynamic` version of the SciMark2 benchmark compared to a refactored version of the original program. The experimental results showed that the execution of method handles in the server VM was 2-5 times slower than native Java invocations [14].

The `invokedynamic` opcode has been used in different language implementations to improve their runtime performance, avoiding the use of reflection. Since its publication in July 2011, there are languages such as JRuby and Groovy that have included the new opcode; others, such as Jython and Rhino [19], are still in process of including it. JRuby is the Ruby implementation for the Java platform. Its version 1.7 supports `invokedynamic`, obtaining a significant performance improvement [34]. However, current released versions of OpenJDK 7 sometimes error out or fail to optimize code as expected [34]. In order to provide a consistent JRuby experience, the use of `invokedynamic` is disabled by default in Java 7. However, the use of `invokedynamic` is enabled by default when running on OpenJDK 8 builds.

Groovy is a dynamic language created to be run on the Java platform. Groovy 2.0 supports `invokedynamic`. Groovy, similar to JRuby, by default disables the use of the

new opcode, giving the user the possibility to activate it. Groovy conducted some tests on version 2.0. The results indicated that runtime performance was increased in some scenarios, while other programs shower slower execution [16].

Jython is an implementation of the Python language for the Java platform. Although this language does not offer full support of the invokedynamic opcode, developers are working on restructuring its code to improve its runtime performance using invoke-dynamic [2]. Shashank Bharadwaj presented a preliminary assessment of the opcode performed on the latest version of Jython. This evaluation showed a 4% improvement across the suite of the evaluated benchmarks [3].

invokedynamic is also being applied to newly created dynamic languages such as Nashorn, a project to be launch for the Java 8 platform. Nashorn is a new implementation of JavaScript [8] that uses the invokedynamic opcode. Nashorn is intended to replace Rhino (the Mozilla implementation of JavaScript for Java platform), which requires a significant rewrite to take advantage of the JSR-292 invokedynamic opcode [36].

Apart from dynamic languages, there are other tools that allow the use of the JVM invokedynamic instruction for different purposes. JooFlux is a JVM agent that provides the dynamic replacement of application aspects and method implementations [26]. invokedynamic allows relinking the method associated to a call site at runtime, a behavior that can be used to implement dynamic aspect-oriented weavers [25]. JooFlux has been compared with Clojure, JRuby, Groovy, JavaScript and Jython, obtaining significant runtime performance benefits [26].

Dynamate is a framework designed to allow the Java implementation of different types of method dispatch, such as multiple dispatch (multi-methods) and the late binding dispatch implemented by dynamic languages [9]. They use invokedynamic to separate method invocation from method dispatch, mapping a different method dispatch technique when a method is called, depending on the desired behavior. Dynamate has been used to prototypically reimplement the JRuby, Jython and Groovy dynamic languages, and the MultiJava, JCop and JastAdd systems [9].

Soot is a Java optimization framework for static analysis and transformation of Java programs, which has been recently extended to support the invokedynamic opcode [4]. Soot provides the Jimple intermediate representation of JVM applications, including invokedynamic. Therefore, Jimple can be used to perform bytecode analysis and optimizations, transforming existing code to invokedynamic; it can also be used to generate invokedynamic instructions with an abstraction level higher than code manipulation tools such as ASM [4].

jDart is a compiler that takes Dart [13] programs and generates binary JVM code to be executed in the Java 7 platform [13]. jDart generates an invokedynamic instruction for every method invocation (except constructors), postponing the method linkage until runtime. A bootstrap included in the language runtime searches for the appropriate method and, once it is linked, the JVM applies the usual optimizations performed for common statically typed code. This language implementation shows the simplicity of using invokedynamic in the implementation of efficient dynamic languages.

6. Conclusions

Dynamic code generation is a suitable technique to build an efficient library that supports the new JVM `invokedynamic` opcode from any high-level language running on the Java platform. In those scenarios where `invokedynamic` may be appropriate, JINDY frees the programmer from generating code that uses this new opcode. A typical use is as an alternative to the reflection API. For this task, JINDY provides a set of services similar to the reflection library. The average performance benefit obtained for this scenario is 14.83 factors for long-running applications, and 9.73 for short-running programs. The library also offers the possibility of avoiding type conversions, reducing the type casts performed with the reflective approach. In this case, the performance benefit increases to 19.97 and 11.97 factors for long- and short-running applications, respectively.

JINDY also allows the developer to provide a custom mechanism to select the method to be invoked, and replaced it at runtime. This alternative is more versatile, making JINDY suitable for in any scenario where the `invokedynamic` opcode can be employed. Compared to reflection, this alternative has an average benefit of 618.53% for long-running applications and 565.42% for short-running programs.

We have also used JINDY to optimize a real application. The OVal validation framework has been re-implemented using our library instead of reflection. For the first constraint validation, our library performs the same as reflection. However, if the number of validations is increased, the performance benefit of the overall application grows up to 79%.

The dynamic code generation technique implemented by JINDY involves a runtime performance and memory consumption penalty. In our computer, we have measured a constant runtime performance cost of 22,068.38 microseconds. When the number of invocations increases, the runtime performance penalty is lower than 1%. The average memory consumption is 3.02%, comparing JINDY with `invokedynamic`.

We plan to use JINDY for the optimization of more real applications that make extensive use of reflection. We will replace the reflective calls with invocations to the JINDY services, and measure runtime performance and memory consumption. We are also considering using JINDY to optimize the implementation of existing dynamic languages that use reflection to provide its dynamically typed services.

The JINDY library, its source code, documentation, and the examples and benchmarks included in this paper can be downloaded from: `http://www.reflection.uniovi.es/invokedynamic/Jindy`

Acknowledgments. This work was funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978, entitled Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages.

References

1. Apache: Log4j, Logging Services (2013), `http://logging.apache.org/log4j`
2. Baker, J.: Making Jython Faster and Better. PYCOM (2012), `https://us.pycon.org/2012/schedule/presentation/446/`

3. Bharadwaj, S.: Invokedynamic Jython. JVM Language Summit (2011), http://wiki.jvmlangsummit.com/Invokedynamic_Jython
4. Bodden, E.: InvokeDynamic support in Soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. pp. 51–55. SOAP '12, ACM, New York, NY, USA (2012)
5. Bruneton, E.: ASM 4.0 A Java bytecode engineering library (September 2011), http://download.forge.objectweb.org/asm/asm4-guide.pdf
6. Chambers, C.: Object-Oriented Multi-Methods in Cecil. In: In ECOOP '92 Conference Proceedings. pp. 33–56. Springer-Verlag (1992)
7. Conde, P., Ortin, F.: The JINDY reference documentation (2012), http://www.reflection.uniovi.es/invokedynamic/doc
8. ECMA-262: ECMAScript Language Specification 5th Edition. European Computer Manufacturers Association (2009)
9. Erhard, K.: Dynamate: A Framework for Method Dispatch using invokedynamic. Master's thesis, Technische Universitat Darmstadt, Mornewegstrabe 30, 64293 Darmstadt, Germany (2012)
10. Fowler, M.: Inversion of control containers and the dependency injection pattern (2004), http://www.martinfowler.com/articles/injection.html
11. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. pp. 57–76. OOPSLA '07, ACM, New York, NY, USA (2007)
12. Hugunin, J.: The Jython Project (2012), http://www.jython.org
13. jDart: A Dart to JVM bytecode compiler, http://code.google.com/p/jdart
14. Kaewkasi, C.: Towards performance measurements for the Java Virtual Machine's invokedynamic. In: Virtual Machines and Intermediate Languages. pp. 3:1–3:6. VMIL '10, ACM, New York, NY, USA (2010)
15. Koenig, D., Glover, A., King, P., Laforge, G., Skeet, J.: Groovy in Action. Manning Publications Co., Greenwich, CT, USA (2007)
16. Laforge, G.: What's new in Groovy 2.0? (Jun 2012), http://www.infoq.com/articles/new-groovy-20
17. Lilja, D.J.: Measuring computer performance: a practitioner's guide. Cambridge University Press, Cambridge, New York, Merlbourne (2000)
18. Microsoft: Windows management instrumentation. Microsoft Developer Network (MSDN) (2012), http://openjdk.java.net/projects/mlvm
19. Mozilla Foundation: Rhino: JavaScript for Java (2012), http://www.mozilla.org/rhino
20. Nutter, C.: The JRuby Community (2012), http://jruby.org
21. Oracle: Java virtual machine support for non-Java languages. Java SE Documentation (2012), http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.htm
22. Oracle: package java.lang.invoke. Java Platform Standard Edition 1.7 (2012), http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html
23. Ortin, F., Cueva, J.M.: Dynamic adaptation of application aspects. Journal of Systems and Software 71(3), 229–243 (May 2004)
24. Ortin, F., Redondo, J.M., García Perez-Schofield, J.B.: Efficient virtual machine support of runtime structural reflection. Science of Computer Programming 74(10), 836–860 (Aug 2009)
25. Ortin, F., Vinuesa, L., Felix, J.M.: The DSAW Aspect-Oriented Software Development Platform. International Journal of Software Engineering and Knowledge Engineering 21(7), 891–929 (2011)
26. Ponge, J., Mouël, F.L.: Jooflux: Hijacking java 7 invokedynamic to support live code modifications. ArXiv, Computer Science, Operating Systems (CoRR) abs/1210.1039 (2012)

27. Redondo, J.M., Ortin, F.: Efficient support of dynamic inheritance for class- and prototype-based languages. Journal of Systems and Software 86(2), 278–301 (Feb 2013)
28. Rose, J.R.: Bytecodes meet combinators: invokedynamic on the JVM. In: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages. pp. 2:1–2:11. VMIL '09, ACM, New York, NY, USA (2009)
29. Spring Framework: XML Schema-based configuration (2013), http://static.springsource.org/spring/docs/2.5.3/reference/xsd-config.html
30. Sun Microsystems Inc.: JSR 292: Supporting Dynamically Typed Languages on the Java Platform (August 2008), http://jcp.org/en/jsr/detail?id=292
31. Sun Microsystems, Inc.: The Da Vinci Machine Project: a multi-language renaissance for the Java Virtual Machine architecture (2008), http://openjdk.java.net/projects/mlvm/
32. Szegedi, A.: Dynalink, an invokedynamic-based high-level linking and metaobject protocol library (2013), https://github.com/szegedi/dynalink
33. Thalinger, C., Rose, J.: Optimizing invokedynamic. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java. pp. 1–9. PPPJ '10, ACM, New York, NY, USA (2010)
34. The JRuby Community: Performance tuning (2012), https://github.com/jruby/jruby/wiki/PerformanceTuning
35. The OVal Development Team: OVal, object validation framework for Java (2013), http://oval.sourceforge.net
36. Traversat, B.: HTML5/JavaScript and Java: The Facts and the Myths. Oracle. JavaOne (2011), http://www.oracle.com/javaone/lad-en/session-presentations/clientside/24821-enok-1439095.pdf
37. Windows Server Techcenter: Windows performance monitor. MicrosoftTechnet (2012), http://technet.microsoft.com/en-us/library/cc749249.aspx

Patricia Conde is a PhD student in cognitive computing and computing with perceptions, at the European Centre for Soft Computing. She is an Engineer in Computer Science and MSc in Web Engineering from the University of Oviedo. Her main research interests are the application of soft computing techniques in the areas of language and cognition applied to control systems. Contact her at patricia.conde@softcomputing.es

Francisco Ortin is an associate professor of the Department of Computer Science at the University of Oviedo, Spain. He is a Computer Science Engineer, and in 2002 he was awarded his PhD. His main research interests include dynamic languages, type systems, computational reflection, and runtime adaptable applications. He is the head of the Computational Reflection research group. Contact him at http://www.di.uniovi.es/˜ortin

Received: January 29, 2013; Accepted: October 16, 2013.