Efficient Runtime Aspect Weaving for Java Applications

Oscar Rodriguez-Prieto[a], Francisco Ortin[a,*], Donna O'Shea[b]

[a] *University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain*
[b] *Cork Institute of Technology, Department of Computer Science, Rossa Avenue, Bishopstown, Cork, Ireland*

Abstract

Context: The aspect-oriented paradigm is aimed at solving the code scattering and tangling problem, providing new mechanisms to support better separation of concerns. For specific scenarios where high runtime adaptability is an important requirement, dynamic Aspect-Oriented Programming (AOP) represents a useful tool. With dynamic AOP, components and aspects can be woven and unwoven at runtime, enabling applications greater responsiveness when dealing with different or changing requirements. However, this responsiveness typically incurs a cost in terms of runtime performance and memory consumption.

Objective: Build an efficient dynamic aspect weaver for Java that provides the best runtime performance compared to the existing approaches, minimum memory overhead consumption, and similar functionalities to the widespread runtime weavers.

Method: We design and implement weaveJ, a dynamic aspect weaver for Java. This dynamic weaver leverages the `invokedynamic` opcode introduced in Java 7, which allows dynamic relinkage of method and field access. We compare the functionalities of weaveJ with the existing dynamic weavers for Java, and evaluate their runtime performance and memory consumption.

Results: weaveJ shows the best runtime performance for all benchmarks and real applications executed. Method interception with `invokedynamic` is at least 142% faster than the techniques used by the existing runtime weavers. The average cost of dynamic weaving using `invokedynamic` is only 2.2% for short running programs, and 1.5% for long running applications. Moreover, the use of aspects in weaveJ does not imply additional memory consumption.

Conclusion: The dynamic aspect weaver implemented demonstrates that `invokedynamic` is a suitable mechanism to provide efficient runtime aspect weaving for Java applications. Moreover, it supports concurrent and programmatic aspect (un)weaving at any point of execution, a wide set of join points, class and object weaving, and allow aspects to have their own state. Neither the Java language nor the virtual machine needs to be modified.

Keywords: Runtime adaptation, aspect weaving, `invokedynamic`, aspect-oriented programming, Java, runtime performance

1. Introduction

Modularity is one of the objectives in software development. It is aimed at separating the different *concerns* in an application, providing a higher level of abstraction, concern reuse, better legibility of each concern in isolation, and higher software maintainability [1]. Although the existing programming paradigms and languages provide different abstractions to modularize applications, some concerns cannot be easily separated from others [2]. Examples of such concerns are logging, information security, caching and persistence [3]. The code of these cross-cutting concerns is commonly spread out over multiple modules, and tangled with code of other concerns: i.e., the code scattering and tangling problem [4].

As mentioned earlier, some cross-cutting concerns cannot be easily decomposed from the rest of modules using the traditional programming paradigms. For this reason, the aspect-oriented paradigm includes programming techniques to support the *Separation of Concerns* (SoC) principle at the source code level [5]. The code that cannot be modularized with the traditional paradigms is implemented with aspects, which are later woven with the components of the application. In this way, components and aspects modularize the different concerns in an application, overcoming the code scattering and tangling problem [2].

Depending on the requirements of applications, aspects can be woven statically (before running the application), at load time (when the classes are first loaded into memory) or dynamically (in a particular point of execution) [6]. Static and load-time weaving commonly provide a relatively small runtime performance penalty, because the component and aspect code is woven before running the application [7]. AspectJ is a widespread aspect-oriented extension to Java that supports load-time weaving with little runtime performance cost [8].

There are specific scenarios where dynamic weaving represents a helpful tool. For example, dynamic weaving has been used in handling Quality of Service (QoS) requirements in distributed systems [9], managing web cache prefetching [10], implementing adaptable security mechanisms in distributed systems [11], balancing the load of RMI applications [12], and changing the control policy of distributed systems [13]. In these cases, dynamic weavers facilitate the implementation of highly runtime-adaptable systems. However, runtime weaving commonly implies significant performance penalties [7].

To facilitate the efficient implementation of dynamic languages, the new `invokedynamic` instruction was added to the Java 7 Virtual Machine (JVM) [14]. This opcode allows users to define and modify method linkage at runtime, applying the existing hotspot optimizations for the JVM [15]. Since this dynamic method linkage is related

*Corresponding author

Email addresses: `rodriguezoscar@uniovi.es` (Oscar Rodriguez-Prieto), `ortin@uniovi.es` (Francisco Ortin), `donna.oshea@cit.ie` (Donna O'Shea)

URL: `http://www.di.uniovi.es/~ortin` (Francisco Ortin), `http://cs.cit.ie/research-staff.donna-oshea.biography` (Donna O'Shea)

to the way dynamic weaving works, `invokedynamic` has been previously identified as a suitable mechanism to provide dynamic aspect weaving [16, 17, 18] –detailed in Section 6. However, this hypothesis has not been validated with the implementation of a dynamic aspect weaver with a set of join points similar to the existing runtime weavers, and thereby no runtime performance evaluation and comparison with existing systems has been undertaken.

The main contribution of this article is weaveJ, an efficient dynamic aspect weaver for Java using the `invokedynamic` JVM opcode, and the evaluation of its runtime performance and memory consumption compared to the existing tools. Runtime weaving with weaveJ is provided as an API, so that programmers can use it programmatically without the knowledge of aspect-oriented languages such as AspectJ. Unlike other approaches, it supports both class and object weaving, and aspects can have their own state because they can be managed as objects. weaveJ provides a wide set of join points and modifies neither the JVM nor the Java language, so any standard Java language and platform implementation can be used. We have evaluated its runtime performance as the best among the existing dynamic aspect weavers for Java (Section 5). Moreover, it does not consume more memory resources than plain Java applications.

The rest of this paper is structured as follows. Section 2 presents an example showing the features of weaveJ, our aspect weaver. `invokedynamic` is described in Section 3, and Section 4 presents the architecture of weaveJ. Section 5 evaluates and compares the existing dynamic aspect weavers, including weaveJ. Related work is discussed in Section 6, and Section 7 presents the conclusions and future work.

2. Motivating example

The following example illustrates some of the features provided by weaveJ. The first part of the example presents how to adapt a running application programmatically. The second part demonstrates how runtime adaption of any Java class can be achieved with two general purpose aspects, illustrating how these aspects can be developed even after application execution. The source code is freely available at [19].

2.1. Programmatic runtime adaptation of particular classes

Figure 1 shows an example component to be adapted at runtime. It models credit cards with `ID` and `balance` fields and `deposit` and `withdraw` methods. Any class and object could be adapted by weaveJ at runtime, following the POJO (Plain Old Java Object) approach [20]. As shown in Figure 1, `CreditCard` requires no additional class extension, interface implementation or member annotation.

Figure 2 shows another Java class used as an aspect. In particular, the `applyCommission` method charges a commission (`commissionPercentage`) to a credit card, if the amount is lower than a given `minValue`. `applyCommission` receives the credit card, the method intercepted and the parameter passed to the intercepted method. The method is represented with an instance of the `MethodHandle` added in Java 7, which provides significantly better runtime performance than reflection [15].

```
01: public class CreditCard {
02:     private double balance;
03:     private long ID;
04:
05:     public CreditCard(double balance) {
06:         this.balance = balance;
07:         ID = CardManager.ID_COUNTER++;
08:     }
09:
10:     public double deposit(double amount) {
11:         return this.balance += amount;
12:     }
13:
14:     public double withdraw(double amount) {
15:         if (amount <= this.balance) {
16:             this.balance -= amount;
17:             return amount;
18:         }
19:         return 0;
20:     }
21: }
```

Figure 1: Example component to be adapted at runtime.

Figure 3 shows different ways to weave components and aspects. First, two credit cards and one commission aspect are created. Then, by using the `Weaver` class of our API, the `CreditCard.withdraw` method is woven with `applyCommission` in the `commission` aspect instance (lines 4-6). From this point of execution on, all the withdrawals against any credit card will be intercepted by the aspect (*around* method call). Notice that the runtime behavior depends on the dynamic state of the `commission` aspect instance: it applies 2% commissions to amounts below 100. The programmer may apply different types of commissions by using different instances of the same class or changing the state of the existing one, following a common object-oriented approach. This approach is different to the one in AspectJ, where aspect instantiation cannot be done programmatically in a particular point of execution.

The example above showed how to perform dynamic weaving of a component class. It is important to note that weaveJ also allows weaving of single objects. This is shown in the second invocation to `weaveAspectForMethodAround` (lines 11-14 of Figure 3), where only the `card2` instance is woven. More precisely, the `rewardSignificant-Deposit` method in the aspect will be called when a deposit is done with `card2` (for this particular object, 0.5% extra credit is applied for deposits equal or greater than 100,000).

In cases where the developer wants an aspect to be no longer woven, the last two lines in Figure 3 show how to perform dynamic unweaving, returning the component to its original state.

2.2. Runtime adaptation of any class

In this section, we will demonstrate how to profile an application and generate log information during a particular execution interval, using weaveJ. These two profiling

```
01: public class CommissionAspect {
02:     private double commissionPercentage;
03:     private double minValue;
04:
05:     public CommissionAspect(double commissionPercentage, double minValue) {
06:         this.commissionPercentage = commissionPercentage;
07:         this.minValue = minValue;
08:     }
09:
10:     public double applyCommission(MethodHandle mh, CreditCard card,
11:                                   double amount) {
12:         System.out.printf("+ Aspect: applying commission to %s... %b\n", card,
13:                           amount < this.minValue);
14:         double funds = (double) mh.invokeWithArguments(card,
15:                               amount >= this.minValue ? amount :
16:                               amount * (1 + this.commissionPercentage / 100));
17:         return funds > 0 ? amount : 0;
18:     }
19:
20:     private static double reward_over_amount = 100_000;
21:     private static double reward_percentage = 0.5;
22:
23:     public static double rewardSignificantDeposit(MethodHandle mh,
24:                                                   CreditCard card, double amount) {
25:         System.out.printf("+ Aspect: Checking reward for deposit of %s...%b\n",
26:                           card, amount >= reward_over_amount);
27:         return (double)mh.invokeWithArguments(card, amount < reward_over_amount ?
28:                                   amount : amount * (1 + reward_percentage/100));
29:     }
30: }
```

Figure 2: Example class used as an aspect.

and logging aspects are useful if we detect an erroneous behavior or bad performance in a particular point of execution. For this case scenario, dynamic weaving is a suitable tool because it allows adding and removing aspects at runtime [21]. We now show how to implement this behavior with weaveJ.

Figure 4 presents the two new aspects to provide runtime profiling and logging. `TraceAspect` provides two methods to log any method invocation before and / or after its execution[1]. These methods use the *before* and *after* method call interception, common in aspect-oriented programming [8]. Likewise, `ProfilerAspect` can be used to measure the execution time elapsed in a method invocation. Its `profileOperation` method is used for *around* method call interception, as the code in Figure 3.

The GUI application in Figure 5 allows (un)weaving any class in a running application using the two aspects in Figure 4. On the left-hand side, the user writes the name of the package where the components to be adapted are placed. Then, they select the class and member (method or field) to be woven. On the right-hand side of the figure, the user can log or profile the selected member (for logging, they must indicate *before*

[1]Signatures of aspect methods are checked by the aspect weaver. They may contain the original types (e.g., `CreditCard`) or a more general type using polymorphism (e.g., `Object`). The Java method overloading resolution algorithm is used to select the appropriate signature [22].

```
01: CreditCard card1 = new CreditCard(1_000), card2 = new CreditCard(1_000_000);
02: card1.withdraw(50);
03: CommissionAspect commission = new CommissionAspect(2, 100);
04: Pointcut pointcutWithdraw = Weaver.weaveAspectForMethodAround(
05:                     "example.component.CreditCard", "withdraw", commission,
06:                     "applyCommission", double.class, double.class);
07: card1.withdraw(50);
08: card2.withdraw(50);
09: card2.withdraw(200);
10:
11: Pointcut pointcutDeposit = Weaver.weaveAspectForMethodAround(
12:               "example.component.CreditCard", "deposit",
13:               "example.aspect.CommissionAspect", "rewardSignificantDeposit",
14:               new Object[] {card2}, double.class, double.class);
15: card2.deposit(100_000);
16: card2.deposit(100);
17: card1.deposit(100_000);
18:
19: pointcutWithdraw.unweave();
20: pointcutDeposit.unweave();
```

Figure 3: Programmatic runtime weaving.

or *after* method execution). At runtime, when an aspect is woven, the logging and profiling messages in Figure 4 will be displayed. When profiling or logging is no longer necessary, the trash button can be clicked to unweave the selected member, going back to the original application execution.

2.3. Functional features of weaveJ

The previous example has illustrated the main features of weaveJ:

1. Aspects can have their own state. The `commission` aspect in Figure 3 (line 3) considers its state (2% of commission) to apply a charge to all the withdrawals.
2. Object and class weaving. weaveJ allows intercepting methods of one single instance (`card2` in line 14 of Figure 3) –i.e., object weaving– and all the instances of a given class (`CreditCard` in line 5) –i.e., class weaving.
3. Programmatic weaving without language extensions. The code in Figure 3 illustrates how aspect (un)weaving is provided as a simple API, and it does not require learning any language extensions.
4. Thread safety. The profiling and logging application presented in Section 2.2 showed how an application can be concurrently (un)woven while it is running.
5. Standard JVM. weaveJ can be used with any standard implementation of the Java platform, since no modifications to the JVM are made.
6. No coupling between components and aspects. Aspects and components are POJOs (Figures 1 and 2), requiring no class extension, interface implementation or member annotation. weaveJ imposes no coupling between them, so aspects can be created even after the application started running. Line 13 in Figure 3 shows how aspect classes and methods can be specified with strings (i.e., not with Java `Class<T>` instances) upon weaving, thereby postponing its evaluation until runtime.

```
01: public class TraceAspect {
02:   public static void traceAfter(String methodName, Object object, Object result) {
03:     System.out.printf("> End of the %s method with result %s.", methodName, result);
04:   }
05:   public static void traceBefore(String methodName, Object object, Object parameter) {
06:     System.out.printf("> Begining of the %s method with object %s and param %s.",
07:                       methodName, object, parameter);
08:   }
09: }
```

```
01: public class ProfilerAspect {
02:   public static Object profileOperation(String methodName, MethodHandle mh,
03:                                         Object object, Object parameter) {
04:     long timeBefore = System.nanoTime();
05:     Object result = mh.invokeWithArguments(object, parameter);
06:     System.out.printf("> Elapsed time invoking %s against the %s object: %d nanos.",
07:                       methodName, object, System.nanoTime() - timeBefore);
08:     return result;
09:   }
10: }
```

Figure 4: Tracing and logging aspects.

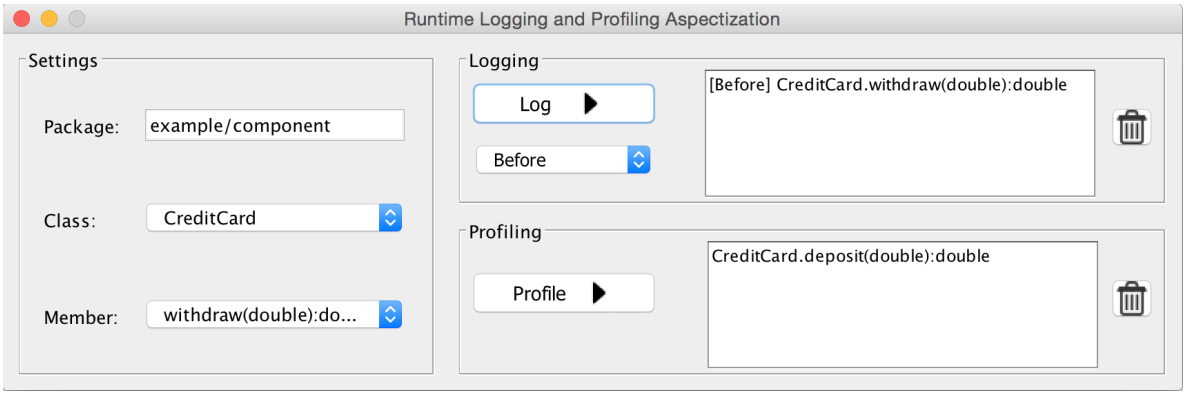

Figure 5: Runtime weaving of any component.

7. Wide set of join points. Although the previous example only used *before*, *after* and *around* method interception, weaveJ allows the interception of other nine join points (see Section 5).

3. The `invokedynamic` instruction

The `invokedynamic` JVM opcode provides a user-defined dynamic linkage mechanism, postponing method and field access resolution until runtime. As shown in the `Application` class in Figure 6, an `invokedynamic` instruction is initially in an unlinked state, meaning that the actual method to be called is unknown at compile time. An `invokedynamic` instruction indicates the method responsible for conducting the dynamic method selection (i.e., the `link` method of the `Bootstrap` class in the center of Figure 6), called *bootstrap method* in the Java documentation [23].

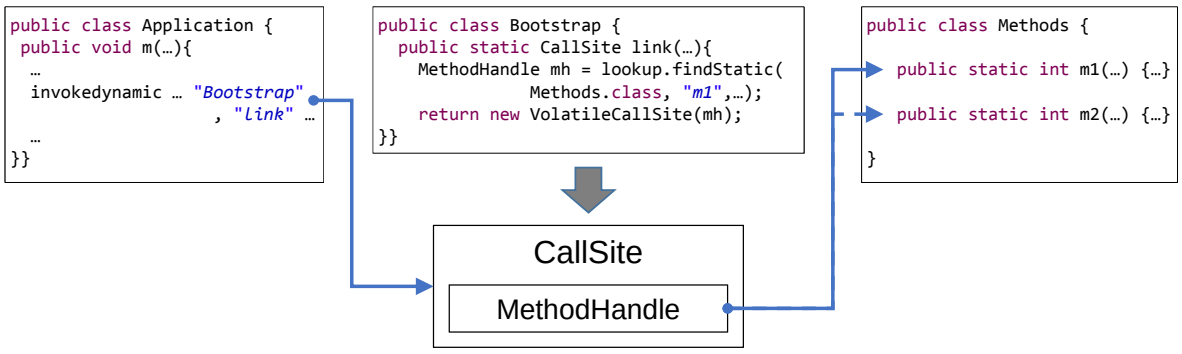

Figure 6: Runtime linkage with `invokedynamic`.

An `invokedynamic` instruction is linked upon its first execution. At runtime, the JVM calls the bootstrap method, which performs the dynamic method resolution. It returns a `CallSite` holding the `MethodHandle` representing the selected method (`m1` in the `Methods` class). In this point of execution, the `invokedynamic` instruction is considered linked. From this point of execution on, the following invocations simply call the method referenced by the returned `MethodHandle`. Moreover, the JVM applies the usual optimizations performed for common statically typed method invocations.

A bootstrap method returns one implementation of the `CallSite` abstract class. The standard Java `invoke` package offers three `CallSite` implementations: `ConstantCallSite` for permanent method handles that cannot be relinked, and `VolatileCallSite` and `MutableCallSite` if relinking is necessary –the main difference between these two classes is that `VolatileCallSite` supports multi-threading. When we want to relink a method because some event occurred at runtime, the `VolatileCallSite` and `MutableCallSite` provide a `setTarget` method to change the `MethodHandle` in the `CallSite`. As shown in Figure 6, this mechanism can be used to change the method invoked in the next execution of the same `invokedynamic` instruction (in our example, calling `m2` instead of `m1`).

4. Architecture

The architecture of weaveJ is presented in Figure 7, showing all its elements and the interaction with the external packages. Figure 8 details its dynamic behavior, showing how components and aspects are dynamically (un)woven.

4.1. Static view

weaveJ implements a Java agent to provide class instrumentation at load time. When the JVM is about to load a class file into memory, the Java agent modifies its assembly code (Figure 7). Particularly, weaveJ replaces eight statically typed opcodes, `invoke`{`virtual`, `interface`, `static`, `special`} and {`get`, `set`}{`field`, `static`}, with the appropriate `invokedynamic` instruction. This code instrumentation has been implemented using the OW2 ASM bytecode manipulation framework [24] and the JINDY library [15].

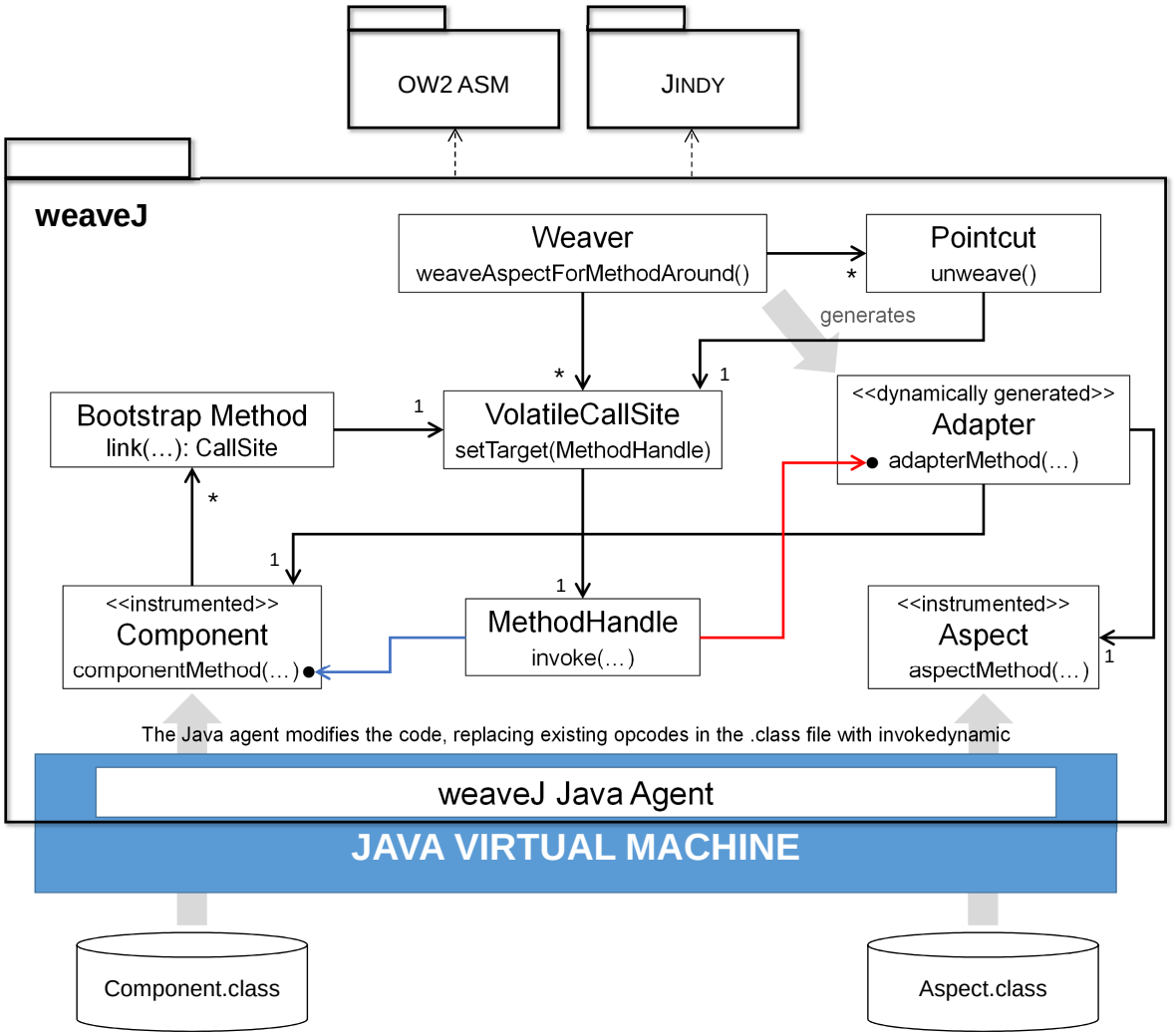

Figure 7: Architecture of weaveJ.

The purpose of using `invokedynamic` is to allow the later adaptation of applications without stopping their execution. To this aim, weaveJ dynamically provides different bootstrap methods that resolve each `invokedynamic` instruction with the original method used in the application. That is, the `invokedynamic` instructions will call the same methods as the original code, but they will allow the later relinkage of methods at runtime to provide method interception. Once the link is established, the JVM performs different hotspot optimizations to provide a runtime performance close to the original opcodes [15].

For each bootstrap method, weaveJ uses one `VolatileCallSite` to allow the concurrent relinkage of the associated method or field at runtime. Figure 7 shows how the bootstrap methods return one `VolatileCallSite` wrapping a `MethodHandle` pointing to the original method in the component (blue arrow). The bootstrap method also registers the returned `CallSite` in the `Weaver` class to allow the later adaptation of the

associated `MethodHandle`.

When the application is running, it could be adapted without stopping its execution. At runtime, a new aspect could be implemented, compiled and placed in the application directory (or in the JVM classpath). By using the `Weaver` class, the compiled aspect can be loaded dynamically as shown in line 13 of Figure 3. Once loaded, the aspect can be used to adapt any running component, using the services provided by the `Weaver` class. It is not necessary to foresee which parts of the application will need to be adapted: any `invokedynamic` instruction could be intercepted.

The `Weaver` class provides different methods for runtime weaving (e.g., `weaveAspectForMethodAround`) by relinking the method with `invokedynamic`. The `invokedynamic` instruction requires the new method to have the same signature as the original one. However, as shown in Figure 4, many aspects require additional information such as the original method and its name. This is the reason why weaveJ dynamically creates an intermediate `Adapter` class to connect the component and the aspect. Using OW2 ASM, our weaver generates the binary code of the `Adapter`. This class simply receives the original parameters and calls the selected aspect, passing additional information. Since the adapter is created dynamically, the specific types of the original method signature are passed (instead of `Object`) avoiding the use of reflection, and hence obtaining better runtime performance [25].

Once the `Adapter` class is generated, the `Weaver` selects the `CallSite` instance associated to the component to be woven, and performs the runtime weaving by changing the method pointed by the `MethodHandle`. In Figure 7, this dynamic weaving process is represented with the replacement of the blue arrow by the red one, causing the invocation to the aspect method via the dynamically generated `Adapter`.

If the method is woven with the *around* behavior, the original method will not be called (the adapter, and hence the aspect, are called instead). However, if the programmer selects either *before* or *after* weaving, the original method must be transparently invoked by weaveJ. For this type of interceptions, the dynamically generated `Adapter` invokes the original component method before or after calling the aspect.

Any aspect could be unwoven at runtime. For that purpose, weaveJ provides `Pointcut` objects, which are returned when components and aspects are woven (lines 4 and 11 in Figure 3). The `unweave` method in `Pointcut` updates its `MethodHandle` to the previous one in the chain of runtime adaptations [6]. Since the particular `Adapter` for an unwoven aspect is no longer necessary, it could be deleted by the JVM garbage collector.

A woven component could be adapted with another aspect. In that case, weaveJ actually weaves the last aspect woven with the component, following the semantics described in [26].

4.2. Dynamic view

Figure 8 shows a dynamic view of the architecture in Figure 7. Particularly, it details how the elements of the architecture are used to provide the runtime weaving caused by part of the code in Figure 3. The following enumeration describes the steps

of the sequence diagram in Figure 8 (each step number in the enumeration is circled in Figure 8).

1. When the main application class is loaded into memory, the weaveJ Java agent is called.
2. That agent
   (a) Instruments the class, replacing eight opcodes with `invokedynamic` (Section 4.1).
   (b) For each `invokedynamic` instruction, an associated bootstrap method is specified.
3. The application is executed (its `main` method is called by the JVM).
4. As in step 2, the `CreditCard` component is instrumented by the agent and loaded into memory.
5. The `withdraw` method in `CreditCard` is called (line 2 in Figure 3).
   (a) Since the `invokedynamic` instruction is not linked yet, the associated bootstrap method is called.
   (b) The bootstrap method creates a `VolatileCallSite`.
   (c) The created `CallSite` wraps a new `MethodHandle` pointing to the `withdraw` method in `CreditCard`.
   (d) The `CallSite` is registered in the `Weaver`, which stores the `CallSite` associated to each method in a hash table.
   (e) The `MethodHandle` in the returned `CallSite` is linked to the `invokedynamic` instruction and executed by the JVM, performing the appropriate optimizations.
6. If `withdraw` is invoked, the linked method will be executed directly (no bootstrap method is called) causing no performance penalty [15].
7. A new `CommissionAspect` is compiled and placed in the application directory.
8. A `weaveAspectForMethod{Around, Before, After}` method of `Weaver` is called (line 4 in Figure 3), indicating the component (`CreditCard`), the aspect (`CommissionAspect`) and the methods to be woven (`withdraw` and `applyCommission`, respectively).
   (a) `CommissionAspect` is loaded into memory and instrumented by the agent.
   (b) An `Adapter` is dynamically generated to connect the component and the aspect.
   (c) The `Weaver` creates a new `MethodHandle` pointing to the adapter method.
   (d) The `Weaver` searches for the `CallSite` associated to `withdraw` in its hash table, and changes the wrapped `MethodHandle` (`setTarget`) to the new one pointing to the `Adapter`.
   (e) A `Pointcut` is created and returned to the caller.
9. After weaving, the `withdraw` method is called (line 7 in Figure 3).
   (a) The JVM invokes the `Adapter` method instead (method interception).

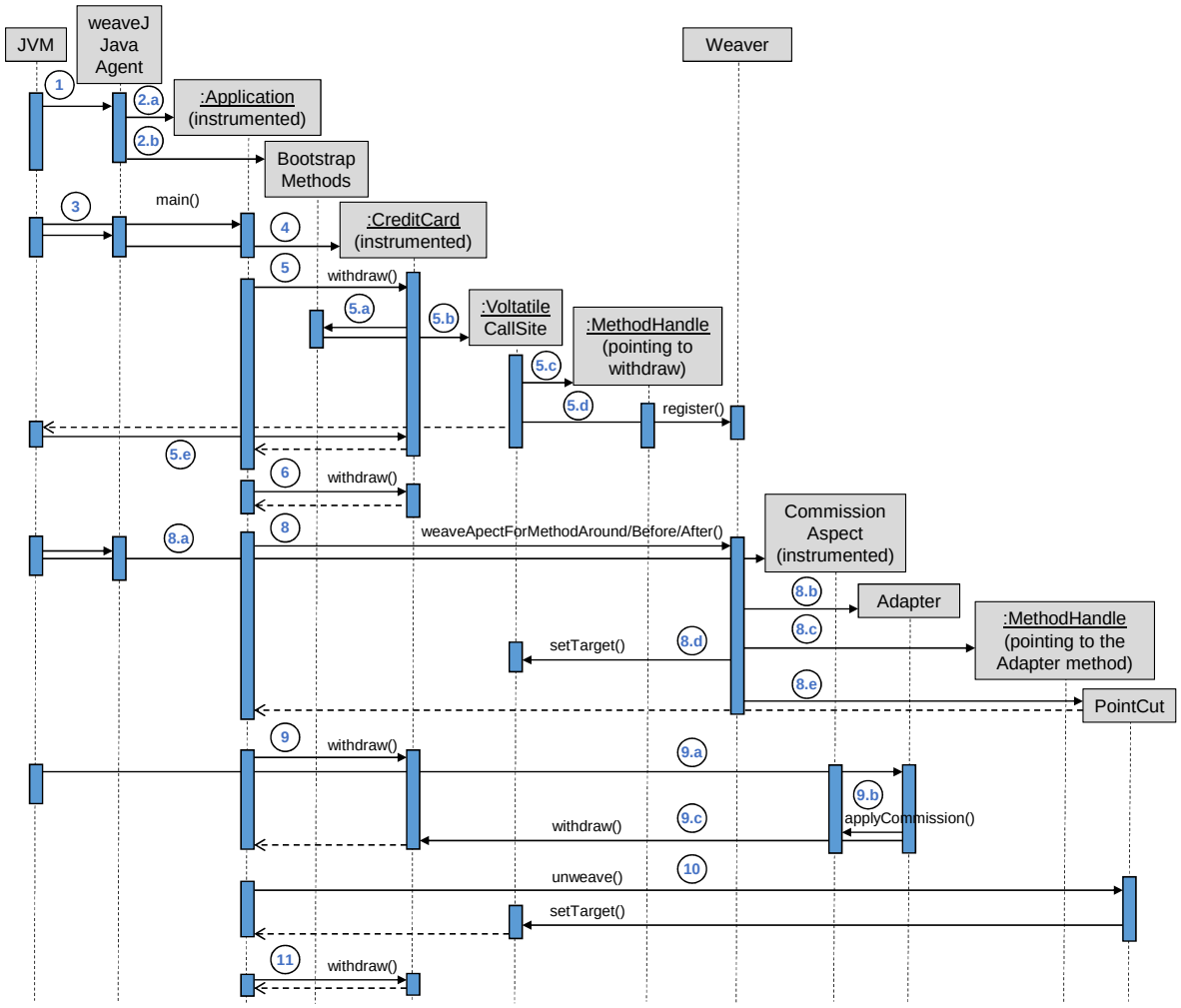

Figure 8: Dynamic view of weaveJ.

(b) The `Adapter` calls `applyCommission` in the aspect (*around* interception).

(c) For *after* interceptions, the original method in the component (`withdraw`) is called by the `Adapter`.

10. If the `unweave` message is passed to the `Pointcut` (line 19, Figure 3), the associated `MethodHandle` is replaced with the original one (`setTarget`).
11. A new invocation to `withdraw` will simply call that method (no interception).

4.3. Complexity of weaveJ

We can discuss the complexity of weaveJ from its dynamic behavior described in the previous subsection. Class instrumentation performed by the Java agent (steps 2, 4 and 8.a) replaces some opcodes with `invokedynamic`. Although the time complexity of this process is linear in the number of instructions in a class file, it only takes place once, when the class is loaded into memory.

In the first execution of an `invokedynamic` instruction (step 5), `CallSite` and `MethodHandle` creation, and bootstrap method invocation are executed in constant time. `CallSite`s are registered in the `Weaver` using a Java `HashMap`, which provides constant time in the average insert (`put`) scenario (when the hash function disperses the elements properly) and linear time in the worst-case scenario. Aspect weaving (step 8) shows the same time complexity, as it uses the very same data structure to get the `CallSite`s.

Once the method is linked, the JVM applies the same optimizations as with direct method invocation (step 6) [15]. Intercepted methods have constant time complexity because the adapter simply performs an invocation to the associated aspect method (step 9). Similarly, the time complexity of unweaving is constant, since both the component and the aspect `MethodHandle`s are stored in the `Pointcut` as references.

Aspect weaving and unweaving have linear space complexity in the number of methods, caused by the hash table used by the `Weaver`. Space complexity for the rest of operations is constant.

5. Evaluation

This section is aimed at evaluating weaveJ and compare it to the existing runtime weavers for Java. The first subsection outlines the experimental methodology used. We then present and discuss runtime performance and memory consumption of the systems analyzed.

5.1. Methodology

5.1.1. Runtime weavers measured

We have studied the existing aspect platforms that support runtime (un)weaving for Java (detailed in Section 6), analyzing whether they provide the features supported by weaveJ (Section 2.3). A summary of that analysis is presented in Table 1, and a detailed discussion is depicted in Section 6. Out of all the platforms analyzed, we evaluate runtime performance and memory consumption of those whose features are close to weaveJ. Particularly, we have selected those that provide programmatic runtime (un)weaving at any point of execution. These are the specific implementations evaluated:

- JBoss AOP 2.1.8.GA [27]. JBoss AOP is a 100% pure Java aspect-oriented framework that can be used in any Java programming environment, or integrated with an application server. It offers a prepackaged set of aspects that can be applied via annotations and pointcut expressions at runtime. Java 6+ is required.

- Spring 4.3.9 [28]. Spring is a layered JEE application framework that supports Aspect-Oriented Programming (AOP). Spring Java runs over Java 6+. It offers an API to add and remove advice at runtime, supporting both load-time and dynamic weaving. Spring Java also supports static weaving through AspectJ.

Feature	JBoss AOP	Spring	JAsCo	PROSE	AABC	JooFlux	JAC	Appeltauer
1	*Partially*	*Yes*	*Yes*	*Yes*	*Partially*	*Yes*	*Yes*	*Yes*
2	*Partially*	*Yes*	*Yes*	*Yes*		*Yes*	*Yes*	*Yes*
3	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*		*Yes*	*Yes*
4	*Yes*	*Partially*	*Yes*		*Yes*	*Yes*	*Partially*	
5	*Yes*	*Yes*	*Partially*	*Yes*				
6	*Yes*	*Yes*					*Yes*	
7	*Partially*	*Yes*	*Partially*	*Partially*			*Yes*	*Yes*
8	*Yes*	*Yes*		*Yes*	*Yes*	*Yes*	*Yes*	
9	18	3	3	5	3	2	2	3

Table 1: Support of the features implemented by weaveJ (features: 1 = runtime weaving at any point of execution; 2 = runtime unweaving at any point of execution; 3 = concurrent (un)weaving; 4 = no coupling between components and aspects; 5 = programmatic (un) weaving; 6 = class and object weaving; 7 = aspects can have their own state; 8 = standard implementation of JVM and Java language; 9 = number of join points).

– JAsCo 0.8.7 [29]. A dynamic AOP language originally tailored for the component-based field. The JAsCo technology allows dynamic integration and removal of aspects with low performance overhead. The JAsCo language is an aspect-oriented extension of Java. It requires a Java 5 compatible virtual machine.

– PROSE 1.4.0 [30]. PROSE (PROgrammable extenSions of sErvices) allows aspect-oriented Java programs to be modified at runtime. PROSE supports dynamic weaving and unweaving of aspects, even if they are unknown at compile time. We have evaluated the PROSE 1.4.0 JVMDI/JVMTI event notification based weaver for the JDK 5.

– weaveJ 1.0. The implementation of our dynamic aspect weaver using the `invokedynamic` opcode for Java 7+ [19].

Besides evaluating the previous aspect weavers, we also implement an object-oriented Java version of each program measured to evaluate the cost of runtime weaving.

5.1.2. Programs measured

We have used different programs to evaluate the efficiency of the selected weavers, including a synthetic micro-benchmark and three real applications [19]:

– Micro-benchmark. We have implemented a synthetic micro-benchmark to measure the cost of join point interception. Table 2 shows how the existing runtime weavers offer a significantly different number of join points. Each cell shows if the *before*, *after* and *around* interceptions are provided. We can see how weaveJ is the second platform in the number of join points supported. The difference between JBoss AOP and weaveJ is that weaveJ does not support method and constructor execution.

  To measure the cost of method interception, the synthetic micro-benchmark evaluates the method call join point provided by all the weavers (for JAsCo, we

	JBoss AOP	Spring AOP	JAsCo	PROSE	weaveJ
Method call	All	All		All	All
Method execution	All		All		
Constructor call	All				All
Constructor execution	All				
Field get	All			Around	All
Field set	All			Around	All
Total	18	3	3	5	12

Table 2: Join points provided by different runtime aspect weavers (*all* represents the support of *before*, *after* and *around*).

measure method execution since that is the only join point supported). The micro-benchmark weaves a method invocation with an aspect that increments an `int` field using *before*, *after* and *around* interceptions. The counter is incremented with one integer parameter passed to the aspect, and the result of the computation is returned. Its source code can be downloaded from [19].

– Mobile communications. This application is based on mobile device communications, where network topologies and communication channels may dynamically change [31]. If the user is connected to a distributed system and it is detected that the communication channel is not secure any more, information is forwarded to a more secure node where transmissions may be encrypted. In this case, an aspect is woven with the distributed application at runtime to tag data for the correct transmission of information through the network [31]. The aspect also forwards the information to the encryption node. Finally, if the mobile device returns to a trusted environment, the aspect is unwoven to avoid any unnecessary overhead. A detailed description of its implementation can be consulted in [31].

– FTP application. We added dynamic aspect weaving to an existing client-server FTP application[2] to cipher all the messages exchanged between the client and the server, when a more secure communication is needed. It is feasible to cipher the channel when critical information is exchanged (e.g. during the client login process), and to use the open channel when the exchanged information is not confidential. In a standard client-server FTP communication, the information is sent and received directly. In an enhanced scenario where ciphering is enabled, all the information passes through a dynamically woven aspect, responsible for encrypting and decrypting data. Before either the server or the client send a FTP command, the aspect encrypts the message; and just after a FTP command is received, the same aspect decrypts it. Thus, the exchanged information travels ciphered using the same channel, transparently to both the client and the server. If the aspect is then unwoven, information travels unencrypted, as it did in the

[2]Apache MINA FTP Server `https://mina.apache.org/ftpserver-project` and JFtp Java Network Browser `http://j-ftp.sourceforge.net`.

original scenario [31].

– JHotDraw. We have extended the well-known JHotDraw GUI framework with runtime aspect weaving using weaveJ [32]. Figures are modeled with components and its representation is implemented with aspects as proposed in [33]. We provide visual and textual representation aspects, which can be dynamically (un)woven without modifying the implementation of the figure components. Logging and profiling is also supported with dynamic weaving, allowing the addition and deletion of these concerns while the application is running (implementation details in [34]).

5.1.3. Data analysis

Measuring execution time of Java programs is not trivial because it is affected by many factors such as just-in-time (JIT) compilation, hotspot dynamic optimizations, thread scheduling and garbage collection. This non-determinism at runtime causes the execution time of Java programs to differ from run to run. In fact, it has been shown how some existing methodologies can be misleading and even lead to wrong conclusions [35]. For this reason, an accurate data analysis must be used to cope with that non-determinism.

We have followed the statistically rigorous methodology proposed by Georges *et al.* [35], which is widely accepted to measure Java applications [36]. That methodology considers two kinds of performances: 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 hot-spot 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 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 computation of the confidence interval is based on the central limit theorem. That theorem states that, for a sufficiently large number of samples (commonly $p \geq 30$), $\overline{x}$ (the arithmetic mean of the x_i measurements) is approximately Gaussian distributed, provided that the samples x_i are independent and they come from the same population [35]. Therefore, taking $p = 30$, we can compute the confidence interval $[c_1, c_2]$ using the *Student's* t-distribution as [37]:

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

Where $\alpha = 0.05$ (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, which follows the *Student's* t-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1-\alpha/2$. 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 [35].

In the start-up methodology, the x_i measurements represent the execution time of the whole process. Therefore, this evaluation method includes the execution time of class loading and instrumentation, helping to analyze its influence on short-running applications. In contrast, the steady-state methodology only considers the execution time of the benchmark, not the whole process. 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 this 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 computer over the $\overline{x_i}$ measurements.

5.1.4. Data measurement

To measure execution time of each benchmark invocation (steady-state methodology), 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 operating system Performance and Reliability Monitor [38]. We measure the difference between the beginning and the end of each benchmark invocation to obtain the execution time of each benchmark run.

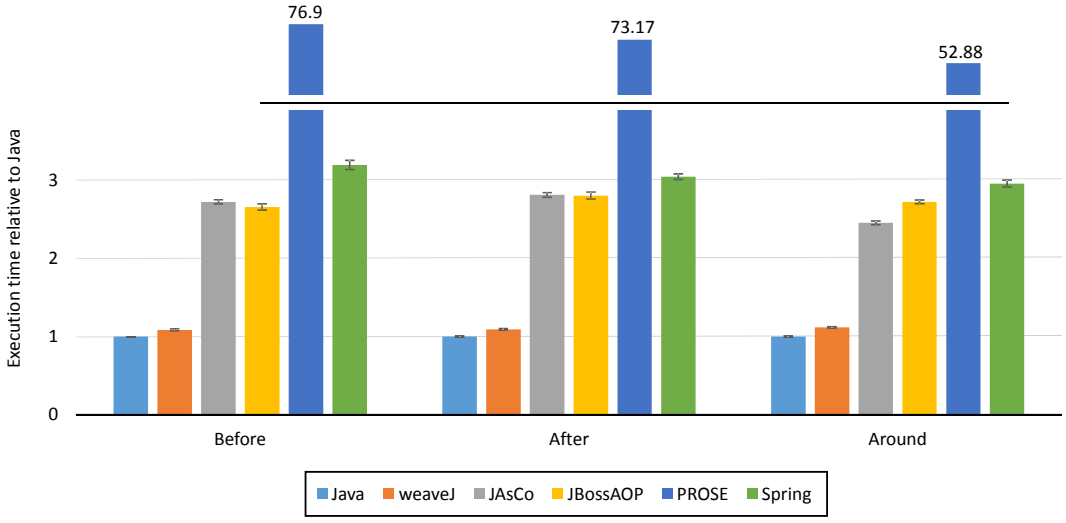

Figure 9: Start-up execution times for the micro-benchmark, relative to Java (whiskers represent the 95% confidence interval).

Memory consumption is measured following the start-up methodology, determining 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) [39]. The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. The `PeakWorkingSet` is measured with explicit calls to the services of the Windows Management Instrumentation infrastructure [40].

All the tests were carried out on a 3.6 GHz Intel I7 4790 system with 16 GB of RAM running an updated 64-bit version of Windows 10.0.14393 Professional. We used Java 8 update 121 for Windows 64 bits. The benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded (until the CPU usage falls below 2% and remains at this level for 30 seconds).

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.

5.2. Runtime performance

We start measuring execution time of the method call join point (method execution in JAsCo). For this purpose, an intercepted method is invoked 10 million times. Its execution is measured with *before*, *after* and *around* interceptions. Since we only measure method invocation, the cost of runtime weaving and unweaving is not considered here. The objective of this micro-benchmark is to evaluate the efficiency of the technique used to provide method interception in isolation, so that we can compare our technique

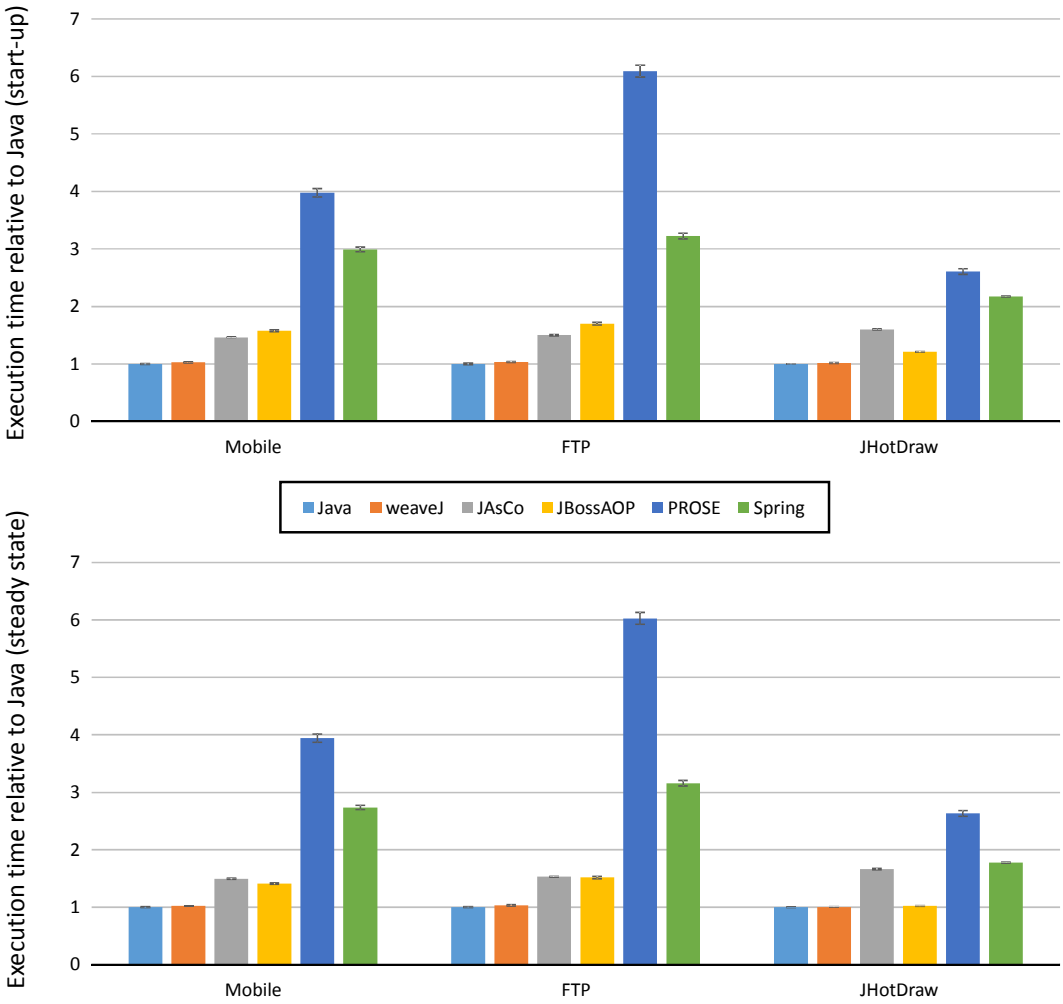

Figure 10: Start-up and steady-state execution times for the three real applications, relative to Java.

with existing approaches. Afterwards, we measure more realistic workloads with real applications where the cost of runtime (un)weaving is considered.

Figure 9 shows the execution time of method call interception for the selected runtime weavers relative to the Java object-oriented implementation (i.e., no aspects). We only show start-up execution times, because the 10 million invocations make the JVM to reach a stable state of execution (steady-state results are essentially the same). We can see how method interception provided by weaveJ is the fastest among all the runtime weavers, for the three different types of interceptions. On average, JAsCo, JBoss and Spring require, respectively, 142%, 148% and 179% more execution time than weaveJ. PROSE is the slowest weaver, requiring 60 times more execution time than our weaver.

The average cost of method interception for weaveJ (i.e., weaveJ vs. Java) is 9.7%; whereas, for the rest of platforms, this cost is at least 165%. As described in Section 4, weaveJ implements method interception with a dynamically generated adapter that receives all the parameters with their specific type (neither reflection nor type casts are

used). Moreover, the JVM keeps optimizing the code when `invokedynamic` is used, so method inlining can achieve a runtime performance similar to the Java implementation.

For JAsCo and JBoss, the average cost of method interception (compared to Java) is 165% and 172%, respectively. They follow the same approach, replacing the component implementation (with a Java agent) to invoke the aspect method. The parameters are wrapped in a general type depending on the kind of interception (e.g., `MethodInvocation` for method execution, `ConstructorInvocation` for constructor execution, etc.). These general types collect the arguments with `Object` references, which must be cast to obtain their original value. This causes a significant performance penalty compared to direct invocation. The additional performance costs imposed by Spring (206%) and PROSE (65 factors) are produced by the interception technique used: Java proxy classes and the JVMDI/JVMTI event notification system, respectively.

Figure 10 shows start-up and steady-state execution times for the three real applications described in Section 5.1.2. In this case, the execution time of weaving and unweaving is included in the measurements. Again, weaveJ is the fastest runtime weaver for the three applications in both methodologies. weaveJ only takes 2.6% more execution time than Java for short-running programs (start-up), and 1.6% for long-running applications (steady state). This low performance cost is due to the way weaveJ performs aspect weaving (Section 4): it simply searches a `CallSite` in a hash table and modifies the associated `MethodHandle`.

With real applications, JBoss is, on average, the second fastest weaver, consuming 44.3% (start-up) and 27.3% (steady state) more execution time than weaveJ. For JAsCo, Spring and PROSE, they require, respectively, 48.7%, 168.9% and 288.2% more execution time than weaveJ for start-up, and 53%, 143.6% and 289% for steady state. JAsCo and JBoss provide aspect weaving by modifying the implementation of methods at runtime using a Java agent. Spring and PROSE basically change the dispatching mechanism to call the method in the aspect instead of the one in the component.

Comparing these results with those for the micro-benchmark, we see that execution times relative to Java are lower in real applications. This is because method interception and aspect (un)weaving are only part of the execution time of the real applications; whereas, in the synthetic micro-benchmark, method interception represents most of the execution time of the whole program.

Steady-state times are lower than start-up for all the weavers, because the JVM performs hot-spot dynamic optimizations when it reaches a steady state [41]. Average improvement in steady state for JAsCo, PROSE and weaveJ are 3.3%, 6.4% and 6.7%, respectively. Spring and JBoss show 17.7% and 20.9% improvement in the steady-state methodology. It seems that the weaving technologies used by JBoss and Spring (Javassist and Java proxy classes, respectively) make it more difficult for the JVM to reach a steady state.

5.3. Memory consumption

Figure 11 shows the memory consumption increase introduced by the different aspect weavers. Spring, weaveJ and JBoss consume 0.20%, 0.24% and 0.47% more memory

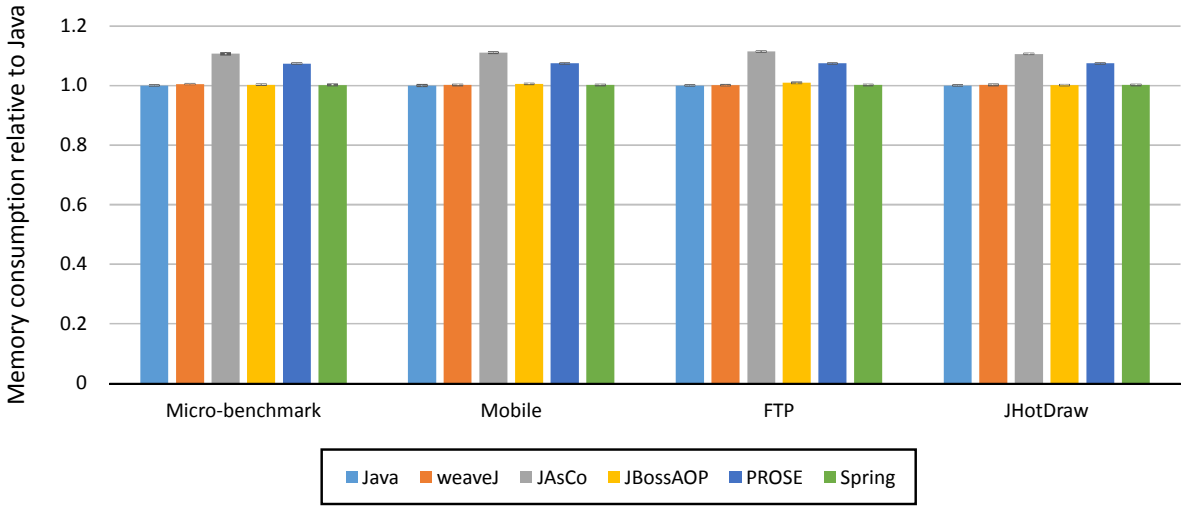

Figure 11: Memory consumption relative to Java.

		Before	After	Around	Mobile	FTP	JHotDraw
Startup	Java	1,325±0.1%	1,324±1.1%	1,283±1.0%	21,313±0.1%	20,123±1.1%	3,547±1.0%
	invokedynamic	1,360±0.9%	1,362±1.5%	1,320±0.9%	21,725±1.0%	20,678±1.2%	3,584±0.8%
	Increase	2.7%	2.8%	2.9%	1.9%	2.8%	1.0%
Steady	Java	1,262±0.2%	1,261±0.7%	1,222±0.6%	20,332±1.0%	19,254±0.7%	3,255±0.6%
	invokedynamic	1,289±0.8%	1,292±1.2%	1,248±0.7%	20,602±0.8%	19,665±1.1%	3,265±0.3%
	Increase	2.2%	2.4%	2.2%	1.3%	2.1%	0.3%

Table 3: Execution time (milliseconds) of `invokedynamic` with no aspects woven (± percentages represent the 95% confidence interval).

than Java. Since the 95% confidence intervals of these three systems overlap, it could not be concluded that differences are statistically significant [35].

PROSE and JAsCo consume more memory than Java. On average, these weavers require 7.39% and 10.95% more memory than the object-oriented application.

5.4. Cost of invokedynamic

As mentioned in Section 4, weaveJ replaces eight bytecodes with `invokedynamic` when classes are loaded into memory. Particularly, we use `VolatileCallSite` to allow the concurrent modification of methods and fields while the application is running. In this section, we measure the cost of using `invokedynamic`. We compare the original Java application with that resulting by replacing the opcodes with `invokedynamic`. Therefore, the difference in execution time and memory consumption indicates the cost of using `invokedynamic` when no aspects are woven.

Table 3 shows the increase in execution time. For start-up, the average cost of `invokedynamic` using `VolatileCallSite` is 2.2%. This value is reduced to 1.5% when the JVM reaches a steady state.

Regarding memory consumption (Table 4), average differences are 0.12%. Since the

	Before	After	Around	Mobile	FTP	JHotDraw
Java	4,454±0.1%	4,470±0.2%	4,491±0.2%	4,476±0.2%	4,488±0.1%	4,351±0.1%
Invokedynamic	4,459±0.2%	4,475±0.1%	4,504±0.1%	4,479±0.1%	4,490±0.1%	4,358±0.1%
Increase	0.11%	0.12%	0.30%	0.06%	0.06%	0.17%

Table 4: Memory consumption (MBs) of `invokedynamic` with no aspects woven (± percentages represent the 95% confidence interval).

95% confidence intervals of memory consumption overlap, it cannot be concluded that differences between Java and `invokedynamic` are statistically significant [35].

6. Related work

Aspect-Aware Bytecode Combinator (AABC) is a Java implementation of an aspect weaver using `invokedynamic` [16]. It is a proof-of-concept prototype to show how *before*, *after* and *around* method call advice could be implemented with that instruction of the JVM. Although `invokedynamic` performs the linkage of methods at runtime, the aspect weaver does not allow aspect weaving in a precise point of execution; it does not allow unweaving either. Particular objects (component instances) cannot be woven, and aspects in AABC are stateless. AABC does not provide the full functionality of a dynamic weaver, and it is still a work in progress [16].

JooFlux is a prototype that shows how `invokedynamic` can be used as a means to provide *before* and *after* method call interception (it does not support *around*) [18]. They implement a Java agent to replace method invocation with `invokedynamic`, but it does not support field access interception. Runtime method call interception is done by using the Java `MethodHandle` combinators to adapt method signatures (weaveJ uses `VolatileCallSite` and dynamic code generation instead). These combinators provide a generic solution with the same signature for every method: parameters are passed as an array of `Objects`, and `Object` is the only return type. This approach is less efficient and robust because expensive type casts need to be done [42], causing a runtime performance cost of between 2.06 and 6.23 factors [18]. JooFlux is not thread safe, and it currently has runtime errors and memory leaks [43]. Object weaving is not provided, and aspects cannot have their own state. In addition, adaptation is not programmatic, and it must be performed through an external JooFlux JMX agent using tools such as jconsole [43].

JAC (Java Aspect Components) is a Java framework for aspect-oriented programming [44]. Unlike most weavers that are class-based, JAC is object-based. It does not require any language extension, providing the features as a Java framework. Aspects can be woven and unwoven at runtime (not programmatically). JAC supports three types of aspect methods: wrappers providing *around* interception of methods and constructors; role methods to add new functionality; and exception handlers. Its design is based on method wrappers and wrapping controllers that resolve aspect composition, contextual operations and wrapping chains at runtime, causing performance issues [44]. Aspects must extend the `Wrapper` class, and they can define their own state.

The `invokedynamic` JVM opcode has been used in the implementation of other runtime adaptable systems. Appeltauer *et al.* built a prototype to support layered method dispatch in context-oriented programming [17]. They compared the performance results with their previous version, obtaining promising results [17]. The Java language is modified to define layers in order to provide *before*, *after* and *around* interception of methods.

Conde and Ortin [15] developed the JINDY library that allows using `invokedynamic` from any language for the Java platform. This library was used to optimize different reflective applications and two dynamic language implementations [45]. Dynamate is a framework that uses `invokedynamic` to support different types of method dispatch, such as multiple dispatch (multi-methods) and the late binding dispatch implemented by dynamic languages [46]. Lagartos *et al.* use `invokedynamic` to provide structural intercession and code generation services for the Java platform [47].

JBoss AOP is an aspect-oriented framework for Java, highly integrated with the JBoss application server [27]. It provides both static and dynamic weaving with a rich set of join points. For dynamic weaving, class files are instrumented statically or at load time using Java agents and Javassist [48]. Programmatic aspect weaving cannot be done within instrumented code. JBoss includes a prepackaged set of aspects such as caching, asynchronous communication, transactions, security and remoting. It supports programmatic weaving, and single instances of components and aspects can be woven. Pointcuts are described in a language similar to AspectJ. Woven aspects cannot been instantiated explicitly and used as common objects.

Spring AOP is an aspect-oriented component included in the Java Spring framework [28]. Aspect-oriented programming can be used to complement Spring Inversion of Control with dynamic aspect weaving. Spring AOP uses standard Java dynamic proxy classes, which allow changing the object that handles a method invocation at runtime [49]. It provides a limited number of join points, but single component instances can be woven. Aspect classes must implement specific interfaces, and they can hold their own state. Spring AOP uses the AspectJ pointcut language definition [50].

JAsCo is an aspect-oriented model for Java that provides dynamic weaving of components and aspect beans (object weaving is not supported) [51]. Aspect beans describe behavior that interferes with the execution of a component by using reusable hooks (special kind of inner classes). Connectors deploy one or more hooks within a specific context [52]. JAsCo does not provide a direct programmatic support for weaving and Java programmers should learn its particular language extensions. Aspects can have state, but they can only be accessed from the connectors (not from the components). Using Java agents, components are replaced by the woven code, providing significant runtime performance optimizations [34].

PROSE (PROgrammable extenSions of sErvices) is an infrastructure that supports dynamic adaptation of applications by providing dynamic aspect weaving at the JVM level [13]. As weaveJ, PROSE does not define a new aspect language, providing the weaving services with a Java API [30]. First versions of PROSE used the JVM debugging interface with expensive performance penalties [53]. They later changed the

implementation with dynamic code replacement, triggering recompilation of methods at runtime [30]. However, as shown in Table 2, PROSE provides a limited number of join points. PROSE does not support weaving of single component and aspect instances, and aspect state cannot be accessed in advice methods [30].

7. Conclusions

We have implemented a dynamic aspect weaver API demonstrating that `invokedynamic` is a suitable mechanism to provide efficient runtime program adaptation, following an aspect-oriented approach. Its runtime performance is significantly higher than the weaving techniques used by the existing dynamic weavers, for all the applications measured. Moreover, the use of aspects does not imply additional memory consumption. This efficiency has been achieved following the next design and implementation strategies:

- The only algorithm in weaveJ that has a linear time complexity is class instrumentation, executed once, when a class is loaded into memory.
- Aspect (un)weaving with `invokedynamic` does not require reloading the whole class at runtime and keeps the hotspot optimizations valid.
- The runtime performance cost of `invokedynamic` compared to the specific JVM opcode goes from 1.5% (steady state) to 2.2% (start-up).
- Aspect weaving is done concurrently, so application execution is not paused.
- Dynamic aspect (un)weaving has a performance cost, but its implementation has a constant time complexity in the average case.
- Method interception is implemented with adapters that are generated at runtime with the specific types of the original method signature, avoiding the expensive use of `Object` and reflection.
- The use of a single data structure to associate call sites to methods produces negligible additional memory consumption.

Besides its efficiency, we have showed how `invokedynamic` can be used to provide a wide set of join points, support both class and object weaving, and allow aspects to have their own state. Runtime weaving and unweaving could be done programmatically at any point of execution. With `invokedynamic`, thread-safe aspect weaving can be provided, changing neither the language nor the virtual machine implementation.

Future work will be focused on adding the method and constructor execution join points to weaveJ. We also plan to add regular expressions to represent the name of different classes, methods and fields to be woven. We will also include a conflict resolution mechanism, which includes semantic conflict correction techniques [54].

The implementation, source code and documentation of weaveJ, and all the benchmarks and examples used in this article are freely available at
`http://www.reflection.uniovi.es/invokedynamic/download/2017/ist`

Acknowledgments

This work has been 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 have also received funds from the Banco Santander through its support to the Campus of International Excellence.

We would like to thank the anonymous reviewers for their detailed list of indications, corrections and suggestions that have helped us to improve the article.

References

[1] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053–1058.

[2] J. M. Felix, F. Ortin, Aspect-oriented programming to improve modularity of object-oriented applications, Journal of Software 9 (9) (2014) 2454–2460.

[3] F. Ortin, J. M. Cueva, Dynamic adaptation of application aspects, Journal of Systems and Software 71 (3) (2004) 229–243.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming, (ECOOP), Springer-Verlag, Berlin, Germany, Finland, 1997, pp. 220–242.

[5] W. Hürsch, C. Lopes, Separation of Concerns, Tech. Rep. NU-CCS-95-03, Northeastern University, Boston (01 1995).

[6] F. Ortin, L. Vinuesa, J. M. Felix, The DSAW aspect-oriented software development platform, International Journal of Software Engineering and Knowledge Engineering 21 (7) (2011) 891–929.

[7] M. Garcia, F. Ortin, D. Llewellyn-Jones, M. Merabti, A performance cost evaluation of aspect weaving, in: Proceedings of the 36th Australasian Computer Science Conference - Volume 135, ACSC, Australian Computer Society, Inc., Darlinghurst, Australia, 2013, pp. 79–85.

[8] Eclipse, The AspectJ Project, `https://eclipse.org/aspectj` (2017).

[9] J. A. Zinky, D. E. Bakken, R. E. Schantz, Architectural support for quality of service for CORBA objects, Theory and Practice of Object Systems 3 (1) (1997) 55–73.

[10] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, J. L. Lawall, Web cache prefetching as an aspect: Towards a dynamic-weaving based solution, in: Proceedings of the 2nd International Conference on Aspect-oriented Software Development, AOSD, ACM, New York, NY, USA, 2003, pp. 110–119.

[11] M. Garcia, D. Llewellyn-Jones, F. Ortin, M. Merabti, Applying dynamic separation of aspects to distributed systems security: A case study, IET Software 6 (3) (2012) 231–248.

[12] A. Stevenson, S. MacDonald, Dynamic aspect-oriented load balancing in Java RMI, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA, 2008, pp. 485–491.

[13] A. Popovici, T. Gross, G. Alonso, Dynamic homogenous AOP with PROSE, Tech. rep., Department of Computer Science, ETH Zürich (2001).

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

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

[16] S. Nopnipa, C. Kaewkasi, Aspect-aware bytecode combinators for a dynamic AOP system with invokedynamic, in: Proceedings of the International Joint Conference on Computer Science and Software Engineering, JCSSE, ACM, 2013, pp. 246–251.

[17] M. Appeltauer, M. Haupt, R. Hirschfeld, Layered method dispatch with invokedynamic: An implementation study, in: Proceedings of the 2nd International Workshop on Context-Oriented Programming, COP, ACM, New York, NY, USA, 2010, pp. 4:1–4:6.

[18] J. Ponge, F. L. Mouël, Jooflux: Hijacking Java 7 invokedynamic to support live code modifications, arXiv:1210.1039 abs/1210.1039.

[19] O. Rodriguez-Prieto, F. Ortin, Efficient runtime aspect weaving with `invokedynamic` – webpage, `http://www.reflection.uniovi.es/invokedynamic/download/2017/ist` (2017).

[20] M. Fowler, POJO, An acronym for: Plain Old Java Object, `https://www.martinfowler.com/bliki/POJO.html` (2000).

[21] J. M. Felix, F. Ortin, Efficient aspect weaver for the .NET platform, IEEE Latin America Transactions 5 (13) (2015) 1534–1541.

[22] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The Java language specification 8th edition, `https://docs.oracle.com/javase/specs/jls/se8/html` (2015).

[23] Oracle, Java Virtual Machine Support for Non-Java Languages, http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html (2017).

[24] OW2, ASM Java bytecode manipulation and analysis framework, http://asm.ow2.org (2017).

[25] F. Ortin, Type inference to optimize a hybrid statically and dynamically typed language, Computer Journal 54 (11) (2011) 1901–1924.

[26] R. Douence, S. Djoko Djoko, P. Fradet, D. Le Botlan, Towards a Common Aspect Semantic Base (CASB) (August 2006).

[27] Red Hat, JBoss AOP, http://jbossaop.jboss.org (2017).

[28] Pivotal, Aspect Oriented Programming with Spring, https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html (2017).

[29] Wim Vanderperren, JAsCo, http://ssel.vub.ac.be/jasco/what.html (2017).

[30] A. Nicoara, G. Alonso, Dynamic AOP with PROSE, in: Advanced Information Systems Engineering, Proceedings of the 17th International Conference, (CAiSE Workshops), 2005, pp. 125–138.

[31] M. Garcia, D. Llewellyn-Jones, F. Ortin, M. Merabti, Applying dynamic separation of aspects to distributed systems security: A case study, IET Software 6 (3) (2012) 231–248.

[32] E. Gamma, T. Eggenschwiler, JHotDraw as open-source project, http://www.jhotdraw.org (2017).

[33] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP, Springer-Verlag, London, UK, UK, 2001, pp. 327–353.

[34] J. M. Felix, A software development platform with both dynamic and static weaving, Ph.D. thesis, Computer Science Department, University of Oviedo (June 2015).

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

[36] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, B. Wiedermann, Wake up and smell the coffee: Evaluation methodology for the 21st century, Communications of the ACM 51 (8) (2008) 83–89.

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

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

[39] 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.

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

[41] 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` (2017).

[42] F. Ortin, J. Quiroga, J. M. Redondo, M. Garcia, Attaining multiple dispatch in widespread object-oriented languages, Dyna 81 (186) (2014) 242–250.

[43] J. Ponge, JooFlux is a Java agent for dynamic aspect-oriented middlewares, `https://github.com/dynamid/jooflux` (2017).

[44] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, JAC: A flexible solution for aspect-oriented programming in Java, in: Metalevel Architectures and Separation of Crosscutting Concerns, Proceedings of the 3rd International Conference Reflection, 2001, pp. 1–24.

[45] 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.

[46] K. Erhard, Dynamate: A Framework for Method Dispatch using invokedynamic, Master's thesis, Technische Universitat Darmstadt, Mornewegstrabe 30, 64293 Darmstadt, Germany (2012).

[47] I. Lagartos, J. M. Redondo, F. Ortin, Towards the integration of metaprogramming services into Java, in: Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE, 2017, pp. 277–284.

[48] S. Chiba, Javassist – a reflection-based programming wizard for Java, in: Proceedings of the ACM OOPSLA Workshop on Reflective Programming in C++ and Java, 1998, pp. 1–5.

[49] Oracle, Dynamic proxy classes, `https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html` (2017).

[50] Eclipse, The AspectJ programming guide, `https://eclipse.org/aspectj/doc/next/progguide/printable.html` (2017).

[51] W. Vanderperren, D. Suvee, Optimizing JAsCo dynamic AOP through HotSwap and Jutta, in: Dynamic Aspects Workshop, 2004, pp. 120–134.

[52] D. Suvée, W. Vanderperren, V. Jonckers, JAsCo: An aspect-oriented approach tailored for component based software development, in: Proceedings of the 2nd International Conference on Aspect-oriented Software Development, AOSD, ACM, New York, NY, USA, 2003, pp. 21–29.

[53] A. Popovici, G. Alonso, T. Gross, Just-in-time aspects: efficient dynamic weaving for Java, in: Proceedings of the Aspect-Oriented Software Development, AOSD, 2003, pp. 100–109.

[54] S. I. Casas, J. B. G. Perez-Schofield, C. A. Marcos, Expert in conflicts, Expert Systems with Applications 36 (3) (2009) 5630–5642.