Efficient runtime aspect weaving for Java applications

Oscar Rodriguez-Prieto\textsuperscript{a}, Francisco Ortin\textsuperscript{*b}, Donna O’Shea\textsuperscript{b}

\textsuperscript{a} Computer Science Department, University of Oviedo, Calvo Sotelo s/n, Oviedo 33007, Spain
\textsuperscript{b} Department of Computer Science, Cork Institute of Technology, Rossa Avenue, Bishopstown, Cork, Ireland

\textbf{ARTICLE INFO}

\textbf{Keywords:}
Runtime adaptation
Aspect weaving
invokedynamic
Aspect-oriented programming
Java
Runtime performance

\textbf{ABSTRACT}

\textit{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.

\textit{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.

\textit{Method:} We design and implement weaveJ, a dynamic aspect weaver for Java. This dynamic weaver leverages the \textit{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.

\textit{Results:} weaveJ shows the best runtime performance for all benchmarks and real applications executed. Method interception with \textit{invokedynamic} is at least 142\% faster than the techniques used by the existing runtime weavers. The average cost of dynamic weaving using \textit{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.

\textit{Conclusion:} The dynamic aspect weaver implemented demonstrates that \textit{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.

1. Introduction

Modularity is one of the objectives in software development. It is aimed at separating the different \textbf{concerns} in an application, providing a higher level of abstraction, concern reuse, better legibility of each concern in isolation, and higher software maintainability \cite{1}. Although the existing programming paradigms and languages provide different abstractions to modularize applications, some concerns cannot be easily separated from others \cite{2}. Examples of such concerns are logging, information security, caching and persistence \cite{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 \cite{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 \textit{Separation of Concerns} (SoC) principle at the source code level \cite{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 \cite{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) \cite{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 to the way dynamic weaving works, invokedynamic has been previously identified as a suitable mechanism to provide dynamic aspect weaving [16–18]—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

Fig. 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 Fig. 1, CreditCard requires no additional class extension, interface implementation or member annotation.

Fig. 2 shows another Java class used as an aspect. In particular, the applyCommission method charges a commission (Commission–Percentage) 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 [19].

Fig. 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 Fig. 3), where only the card2 instance is woven. More precisely, the rewardSignificantDeposit 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 Fig. 3 show how to perform dynamic unwaving, 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 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.

Fig. 4 presents the two new aspects to provide runtime profiling and logging. TraceAspect provides two methods to log any method
دریافت فوری متن کامل مقاله

امکان دانلود نسخه تمام متن مقالات انگلیسی
امکان دانلود نسخه ترجمه شده مقالات
پذیرش سفارش ترجمه تخصصی
امکان جستجو در آرشیو جامعی از صدها موضوع و هزاران مقاله
امکان دانلود رایگان ۲ صفحه اول هر مقاله
امکان پرداخت اینترنتی با کلیه کارت های عضو شتاب
دانلود فوری مقاله پس از پرداخت آنلاین
پشتیبانی کامل خرید با بهره مندی از سیستم هوشمند رهگیری سفارشات