INVESTIGACIÓN & DESARROLLO, No. **11**, Vol. **1**: 122 – 136 (2011)
ISSN 2518-4431

# MAJOR: AN ASPECT WEAVER WITH FULL COVERAGE SUPPORT
# MAJOR: UN *ASPECT WEAVER* CON SOPORTE PARA COBERTURA TOTAL

**Alex Villazón*, Walter Binder**, Philippe Moret** and Danilo Ansaloni****
******Centro de Investigaciones de Nuevas Tecnologías Informáticas* – CINTI
*Universidad Privada Boliviana, Bolivia*
***Faculty of Informatics, University of Lugano, Switzerland*
avillazon@upb.edu
(Recibido el 23 de octubre 2011, aceptado para publicación el 13 de diciembre 2011)

## ABSTRACT

Prevailing Aspect-Oriented Programming (AOP) frameworks for Java, such as AspectJ, use bytecode instrumentation techniques to weave aspects into application code. Unfortunately, those frameworks do not support weaving in the Java class library. When implementing aspect-based tools, such as profilers, debugger, or dynamic program analysis tools in general, the aforementioned restriction becomes an important limitation for successfully applying AOP.
In this article we present MAJOR, an aspect weaver with full coverage support. That is, MAJOR ensures that aspects are woven into all classes loaded in a Java Virtual Machine, including those in the standard Java class library. We describe the intricacies of instrumentation of the Java class library and present an extended instrumentation approach allowing the user to choose between a pure Java weaving solution based on a two-phases instrumentation scheme, or a single-phase one requiring a tiny native code layer. The single-phase approach allows to better isolate the weaving process from the execution of the woven code.

## RESUMEN

Los *frameworks* actuales de Programación Orientada a Aspectos (*Aspect-Oriented Programming* - AOP) basados en Java, como ser AspectJ, utilizan técnicas de instrumentación de código de bajo nivel (*bytecode*) para insertar o aplicar aspectos dentro del código de las aplicaciones (proceso llamado *aspect weaving*). Lamentablemente, dichos sistemas no cuentan con soporte para aplicar aspectos dentro de la biblioteca estándar de clases de Java (*Java class library*). Esta restricción resulta en una limitación importante para la utilización exitosa de AOP en la implementación de herramientas basadas en aspectos, tales como *profilers*, *debuggers*, o herramientas para análisis de programas dinámicos en general. En este artículo se presenta MAJOR, un *aspect weaver* con soporte para cobertura total. Es decir, MAJOR se asegura que los aspectos son aplicados a todas las clases que son cargadas en una máquina virtual Java (*Java Virtual Machine*), incluyendo aquellas clases de la biblioteca estándar de Java. Se describe la complejidad de la instrumentación de dichas clases y se presenta un nuevo enfoque de instrumentación que permite la selección entre una solución en dos fases que está basada únicamente en Java, o una solución en una sola fase, pero que requiere una pequeña capa de código nativo. La solución monofásica permite aislar de mejor manera el proceso de aplicación de aspecto (*aspect weaving*), de la ejecución del código ya instrumentado con el aspecto (*woven code*).

**Keywords:** Aspect-oriented Programming, Aspect Weaving, Bytecode Instrumentation, Java Virtual Machine.
**Palabras Clave:** Programación Orientada a Aspectos, *Aspect weaving*, Instrumentación de *Bytecode*, *Profiling*, *Debugging*, Máquina Virtual Java.

## 1. INTRODUCTION

Instrumentation and manipulation of Java bytecode have become important techniques for the implementation of various tools and frameworks. For this reason, modern Java Virtual Machine (JVM) offer dedicated instrumentation support, such as the JVM Tool Interface (JVMTI) [1] and through the `java.lang.instrument` API. In addition, there are several instrumentation libraries for Java bytecode, such as BCEL [2], ASM [3], or Javassist [4], as well as for other languages [5]. However, such interfaces and techniques are considered as rather low-level, and even though they have been successfully used to build tools for profiling, debugging, testing, and reverse engineering, the implementation of new tools using them is tedious and error prone. In our work, we explore the use of Aspect-Oriented Programming (AOP) [6] as a way to express instrumentations in a higher-level abstraction for the development of tools, so as to hide low-level instrumentation details from the tool developer, easing maintenance and extension of tools.

AOP enables clean modularization of crosscutting concerns, such as error checking and handling, synchronization, monitoring and logging, and debugging support. AspectJ [7] is the de-facto standard language and framework providing AOP capabilities to Java. AspectJ uses bytecode instrumentation to support code transformations for aspect weaving [8], which is based on the insertion of code at well-defined points in Java programs without resorting to source code manipulation.

**DOI:** 10.23881/idupbo.011.1-8i

Unfortunately, prevailing aspect weavers do not support aspect weaving with full coverage, that is, *instrumentation of any method with a bytecode representation*, notably because they prevent weaving in the Java class library. Hence, the applicability of AOP is limited, because tools such as profilers often require full coverage. For example, when using standard AspectJ weavers, profiling aspects, such as DJProf [9], cannot profile method execution within the Java class library, resulting in incomplete profiles. This issue was reported as one of the main limitations of the approach by the authors of DJProf.

Another issue is that techniques for profiling, debugging, and reverse engineering often benefit from detailed calling context information [10, 11]. As an example in the debugging area, calling context information has been used to reproduce the crashing conditions of a faulty application by storing copies of method arguments on a *shadow stack*, so as to keep information on an eventual crash [11]. Unfortunately, AspectJ does not offer dedicated support for efficiently accessing detailed calling context information. While it is possible to specify aspects to gather calling context information, such aspects typically cause high runtime overhead, thus limiting the applicability of AOP for profiling and debugging. Our work aims at filling this gap by enabling high-level instrumentation with full coverage and efficient access to complete calling context information.

Instrumentation of the standard Java class library is difficult and may crash the JVM, because of its sensitivity to modifications, notably during JVM bootstrapping, for instance by triggering premature initialization of classes used by inserted code. While aspect support can be integrated directly within the JVM [13], such an approach is restrictive because it prevents the reuse of existing AOP tools. In addition, using modified JVMs and native code prevents to leverage standard, state-of-the-art JVM technologies.

In prior work we introduced FERRARI (Framework for Exhaustive Rewriting and Reification with Advanced Runtime Instrumentation) [12], a generic bytecode instrumentation framework supporting the instrumentation of the whole Java class library including all core classes, as well as dynamically loaded classes. FERRARI provides a flexible interface enabling different user-defined instrumentations (UDIs) written in pure Java to control the instrumentation process.

In this article, we present MAJOR[1] a framework for aspect weaving with full coverage support based on FERRARI and related techniques. MAJOR enhances the standard AspectJ weaver, without requiring any modification of the weaver itself. That is, MAJOR applies different transformations to the woven code so as to add support for full coverage and calling context reification. Different aspect-based tools can be implemented using MAJOR, which relies on a special User-Defined Instrumentation for AspectJ (AJ-UDI). MAJOR also includes a module that combines aspect weaving with calling-context reification provided by the Calling Context UDI (CC-UDI). The framework absolves the developer from dealing with low-level instrumentation details, which are handled by the standard AspectJ weaver in conjunction with FERRARI and MAJOR. We have validated our approach with different aspect-based tools that takes advantages of the distinguishing features of MAJOR. For example, in [15] we introduced an aspect for memory leak detection that also benefits from full coverage weaving and we also presented an aspect subsuming the functionality of ReCrash [11], an existing tool based on low-level bytecode instrumentation techniques that generates unit tests to reproduce program failures. Our aspect-based tools are concisely implemented in a few lines of code and leverage MAJOR for aspect weaving with full coverage and for efficient access to calling context information.

This article refines, extends and complements the concepts introduced in [14,15]. As contributions, in this article we highlight the benefits of aspect weaving with full coverage and describe the techniques underlying MAJOR. We discuss an extended approach that expands the instrumentation layer in order to support single-phase weaving for all classes included those of the Java class library, compared to the basic approach based on two-phases weaving scheme. This extension allows to completely separating the weaving process from the execution of woven code. We show that our approach is compatible with the standard AspectJ.

This article is structured as follows. Section 2 summarizes the features provided by FERRARI, our generic instrumentation framework. Section 3 introduces basic concepts of aspect weaving and discusses the intricacies of weaving the Java class library. Section 4 discusses the implementation of the AJ-UDI and how we adapt the AspectJ weaver to support comprehensive aspect weaving. Section 5 discusses the extended mechanism to support weaving in a single-phase weaving scheme. Section 6 addresses related work. Finally, Section 7 concludes this article.

## 2. INSTRUMENTATION FRAMEWORK

Our framework for aspect weaving with full coverage relies on FERRARI [12], a generic bytecode instrumentation framework supporting the instrumentation of all classes loaded in a JVM. FERRARI is neither an application-specific framework nor a low-level bytecode instrumentation toolkit. Instead, it generates the necessary program logic to enable

[1] MAJOR stands for MAJOR is AspectJ with Overall Rewriting

custom instrumentation of the Java class library and of application classes. FERRARI solves the problem of bootstrapping the JVM with an instrumented Java class library. It provides a flexible interface enabling different user-defined instrumentation modules (UDIs), which may be written in pure Java, to control the instrumentation process. Figure 1 shows an overview of the instrumentation architecture.

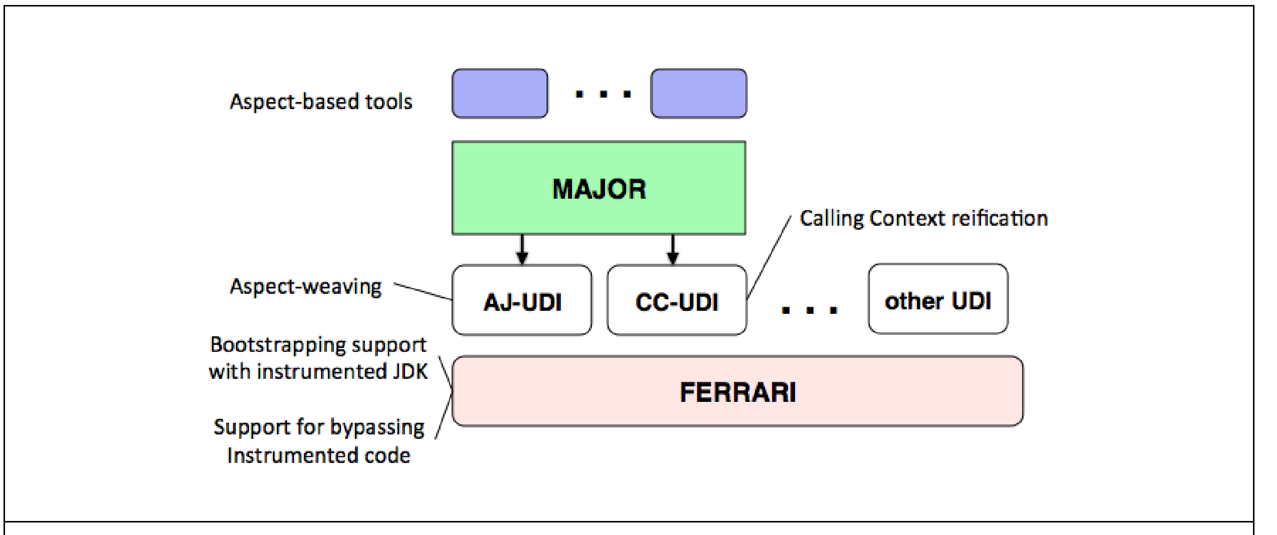
Aspect-based tools
. . .
MAJOR
Calling Context reification
Aspect-weaving
AJ-UDI
CC-UDI
. . .
other UDI
Bootstrapping support with instrumented JDK
FERRARI
Support for bypassing Instrumented code

**Figure 1 –** Architecture Overview.

FERRARI consists of a static instrumentation tool and a runtime instrumentation agent. FERRARI defines an interface that the UDI has to implement and invokes the UDI through this interface. The UDI may change method bodies, add new methods (with minor restrictions), and add fields (with some restrictions). FERRARI passes the original class bytes to the UDI and receives back the UDI-instrumented class bytes. FERRARI's general purpose API [12] allows the seamless integration of existing bytecode transformation tools through UDIs.

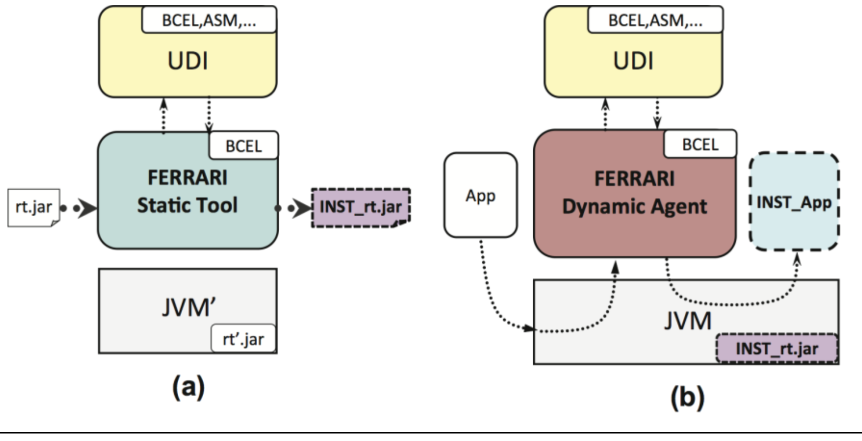

**Figure 2 –** (a) static and (b) dynamic instrumentation with FERRARI.

FERRARI provides a tool to statically instrument the Java class library according to a given UDI. Figure 2(a) shows the static instrumentation of the Java class library (represented by `rt.jar`), resulting in the instrumented version `INST_rt.jar` that is used by the JVM executing the application under dynamic instrumentation (see Figure 2(b)). Application classes are dynamically instrumented by FERRARI's agent in collaboration with the UDI. The agent is based on the `java.lang.instrument` API introduced in Java 5, ensuring portability. Figure 2(b) shows how application classes (`App`) are instrumented dynamically so that their instrumented versions (`INST_App`) are those actually linked by the JVM. As an extension to this base instrumentation scheme, we discuss in Section 5 an approach to instrument the Java class library without resorting to a static tool. This new approach requires the use of a tiny native code layer, but better isolates the weaving process from the execution of woven code.

The current implementation of FERRARI uses Apache's bytecode engineering library BCEL [2]. UDIs are nevertheless free to use any bytecode engineering library, such as BCEL, ASM [3], Javassist [4], Soot [16], etc.

FERRARI offers generic mechanisms to ensure instrumentation with full coverage of any code in a system which has a corresponding bytecode representation. To this end, (1) it ensures that UDI-inserted code is not executed before the JVM has completed bootstrapping and (2) it provides support for temporarily bypassing the execution of inserted code for each thread during load-time instrumentation.

Regarding issue (1), FERRARI keeps a copy of the original code of every instrumented method and uses a global flag, the Bootstrap Inserted-code Bypass (BIB), to bypass the execution of UDI-inserted code during the bootstrapping of the JVM. That is, the bootstrapping lasts until FERRARI's instrumentation agent starts execution, which happens before the end of the JVM startup (see the JVM Specification, Second Edition, Section 5.5) [17].

Concerning issue (2), FERRARI introduces a thread-local flag, the Dynamic Inserted-code Bypass (DIB), which allows per-thread bypassing of UDI-inserted code. To this end, FERRARI inserts the boolean instance field **`dibFlag`** into the **`java.lang.Thread`** class. If the flag is set to true, the current thread bypasses UDI-inserted code. The **`dibFlag`** is exposed to the UDI developer through the DIB class (see Figure 3).

```
public final class DIB {
  public static boolean getFlag() {
    return Thread.currentThread().dibFlag;
  }

  public static void setFlag(boolean value) {
   Thread.currentThread().dibFlag = value;
  }

  public static boolean getFlagAndSetTrue() {
   Thread t = Thread.currentThread();
   boolean value = t.dibFlag;
   t.dibFlag = true;
   return value;
  }

...
}
```

**Figure 3-** API for manipulating the Dynamic Inserted-code Bypass (DIB).

The bypasses induce some constraints for UDI development. UDI-inserted static or instance fields must be initialized to Java's default values. Otherwise, the inserted code to initialize the added fields may be bypassed resulting in incompletely initialized classes or objects. To mitigate this limitation, FERRARI offers special support for introducing extra classes that can hold added static fields [12]. This support is only available for the static instrumentation of the Java class library. In Section 5 we will also discuss the case of extra classes introduced by the UDI when dealing with instrumentation of the Java class library without the static tool.

In general, UDI-inserted code often introduces dependencies on UDI-specific runtime classes (we call them "UDI-runtime-classes"). The UDI developer must ensure that methods in UDI-runtime-classes do not execute any instrumented code. To this end, the DIB mechanism can be used as shown in Figure 4, which allows the calling thread to temporarily bypass UDI-inserted code at runtime. This issue is particularly important in support aspect weaving with full coverage and will be explained in Section 3.2.

```
boolean oldValue = DIB.getFlagAndSetTrue();
try {
  // bypassing UDI-inserted code in invoked methods
  ...
} finally { DIB.setFlag(oldValue); }
```

**Figure 4 -** Activating the DIB for the current thread.

When a new thread is created, it "inherits" the **`dibFlag`** value from the current thread. FERRARI modifies the constructors of **`java.lang.Thread`** accordingly. Consequently, if the UDI or UDI-runtime-classes spawn threads while the **`dibFlag`** is true, the new threads will also bypass UDI-inserted code. By default, the **`dibFlag`** is set to false and therefore instrumented code is executed when the BIB is disabled by FERRARI's agent after bootstrapping. This ensures that UDI-inserted code is executed when the application's **`main(...)`** method is invoked.

FERRARI classes, the instrumentation agent, the UDI, UDI-runtime-classes, and classes of the bytecode engineering library are excluded from normal instrumentation. In order to avoid name-clashes with application classes, these classes are loaded into a special classloader namespace. All other classes are instrumented either statically or dynamically at load-time. In Section 5, we will discuss an extended approach that avoids this explicit namespace separation. That is, the instrumentation is performed in a separate JVM. Thus, there is no perturbation possible between the instrumentation and the execution of the instrumented code. This approach allows load-time instrumentation of all classes, but relies on a tiny native code layer.

## 3. ASPECT WEAVING WITH FULL COVERAGE

To better understand how the support for weaving with full coverage is achieved, we present background information of the most important concepts of AOP and aspect weaving. Then we describe the intricacies of weaving the Java class library.

### 3.1. Aspect-Oriented Programming

In AspectJ, an aspect is an extended class with additional constructs. A *join point* is any identifiable execution point in a system (e.g., method call, method execution, object construction, variable assignment, etc.). Join points are the places where a crosscutting action can be inserted. The user can specify weaving rules to be applied to join points through so-called *pointcuts* and *advice*. A pointcut identifies or captures join points in the program flow, and advice are the actions to be applied.

AspectJ supports three kinds of advice: *before*, *after*, and *around*, which are executed prior, following, or surrounding a join point's execution. The *AspectJ compiler* compiles aspects into standard Java classes. In the aspect class, advice are compiled into methods. During the weaving process, the *AspectJ weaver* inserts code in the woven class to invoke these advice methods. Advice can receive some context information, e.g., to identify which join point has been captured.

During the execution of a woven class, by default, a singleton instance of the aspect is instantiated. Several aspects can be woven simultaneously and can therefore coexist during the execution.

### 3.2. Weaving the Java Class Library

In order to better understand the difficulties of weaving aspects in the Java class library, let us consider a simple profiling aspect. Figure 5 shows the **`ObjectTracker`** aspect that captures every object allocation by intercepting constructor calls (specified by **`call(*.new(..))`**) in the **`allAllocs`** pointcut). After each object allocation, a *weak reference* to the allocated object is stored in a set. In Java, **`WeakReference`** type allows to prevent the referenced object from being garbage collected. The **`ReferenceQueue`** instance is used by the garbage collector to place weak reference that are cleared. Thus, the set **`refs`** must be updated atomically, as a single instance of the aspect is shared by all executing threads. The **`!within()`** pointcut designator prevents the aspect itself to be woven in order to avoid infinite recursions. The aspect may periodically analyze the set **`refs`** and use Java's shutdown hook mechanism to dump the final results upon application termination (not shown in Figure 5).

```
public aspect ObjectTracker {
  static private final Set<WeakReference> refs =
    Collections.synchronizedSet(new HashSet<WeakReference>());
  static private final ReferenceQueue<Object> refQueue = new ReferenceQueue<Object>();

  pointcut allAllocs() : call(*.new(..)) && !within(ObjectTracker);

  after() returning(Object o) : allAllocs() {
    WeakReference objref = new WeakReference(o, refQueue);
    refs.add(objref);
  }
...
}
```

**Figure 5 -** Object allocation tracking using an aspect

The code of the **`ObjectTracker`** aspect could be part of a memory leak profiling tool, where it is essential to keep track of memory allocations also in the Java class library. This is because many memory leaks occur in containers that reference unused data entries [18, 19], and usually the standard Java collection classes are used as containers. Unfortunately, when woven with prevailing AspectJ weavers, the aspect will track only allocations in application code. Consequently, without aspect weaving with full coverage, AOP is not well suited for building sophisticated profiling and debugging tools, such as the container-based heap tracking profiler described in [18]. Our approach aims at filling this gap, while ensuring portability and compatibility with standard AspectJ.

There are two main difficulties for weaving the Java class library. The first one concerns the sensitivity to modifications of core Java classes, notably during JVM bootstrapping. Because the weaving process introduces new dependencies between the woven classes and the aspects, loading a woven core Java class, such as **`java.lang.Object`**, may break bootstrapping and eventually crash the JVM. To solve this issue, we rely on FERRARI's support for bootstrapping the JVM with an instrumented Java class library.

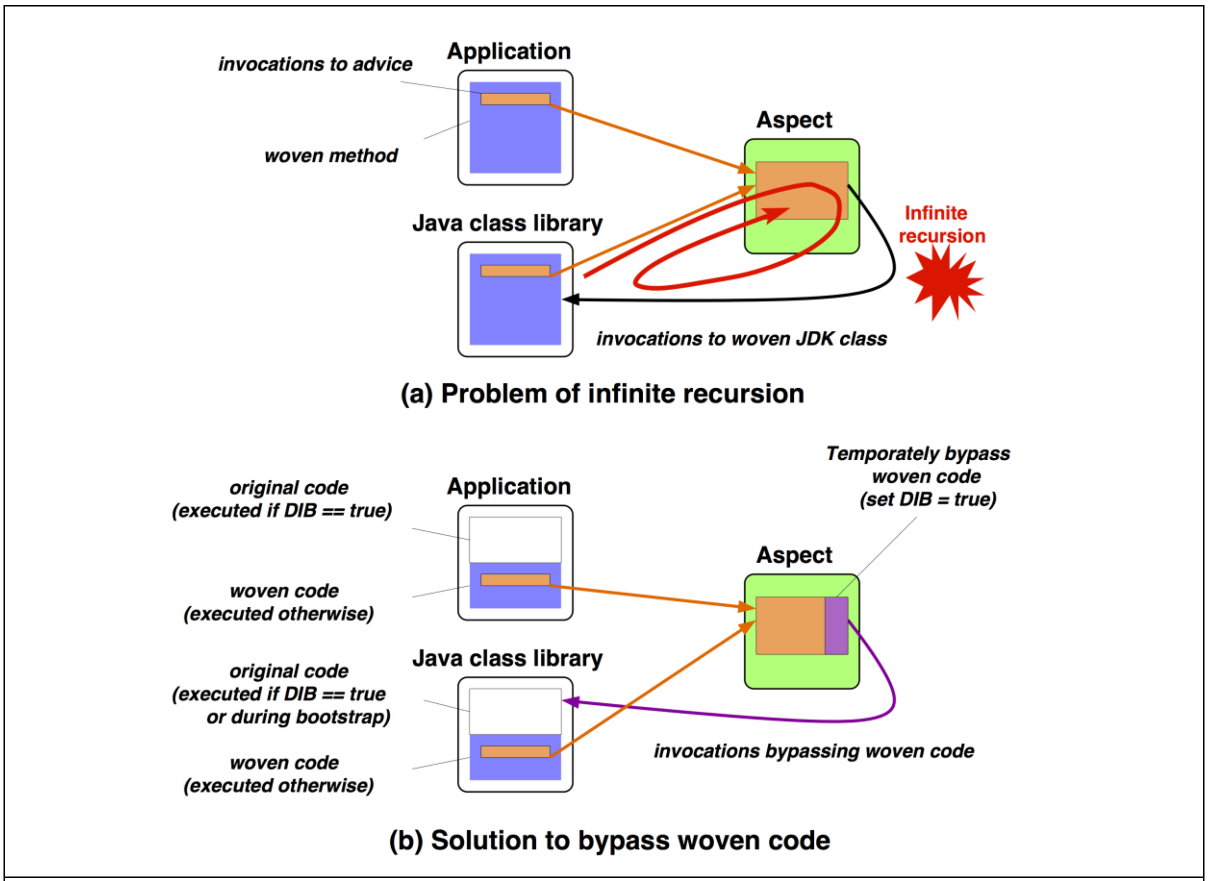
invocations to advice
Application
woven method
Aspect
Java class library
Infinite recursion
invocations to woven JDK class
(a) Problem of infinite recursion
original code (executed if DIB == true)
Application
Temporarily bypass woven code (set DIB = true)
Aspect
woven code (executed otherwise)
Java class library
original code (executed if DIB == true or during bootstrap)
woven code (executed otherwise)
invocations bypassing woven code
(b) Solution to bypass woven code

**Figure 6 –** Avoiding infinite recursions within woven classes in the Java class library

The second one is related to advice execution during the execution of woven code. In the example of the **`ObjectTracker`** aspect in Figure 5, the advice body allocates an instance of the **`WeakReference`** class, which is part of the Java class library. As shown in Figure 6(a), if the aspect is woven into both the application code and the Java class library using prevailing aspect weavers (assuming that the bootstrapping problem is solved), an infinite recursion will happen, because the invocation of woven code by the advice will recursively trigger the invocation of the advice (which captures also the **`WeakReference`** allocation).

To solve this issue, MAJOR uses FERRARI's DIB mechanism. The method bodies are duplicated such that the original and the instrumented versions of the bytecode are kept together, and a conditional selects the version to be executed upon method entry depending on the state of the DIB. This solution is illustrated in Figure 6(b). The pseudo-code in Figure 7 shows how the code pattern to activate the DIB is applied to advice execution.

```
aspect Foo {
  <advice> : <pointcut> {
    boolean oldValue = DIB.getFlagAndSetTrue();
    try {
      // advice body invoking code in a woven Java class library
      ...
    } finally { DIB.setFlag(oldValue); }
  }
}
```

**Figure 7-** Avoiding infinite recursions in woven code within the DIB mechanism.

## 4. THE ASPECTJ UDI (AJ-UDI)

MAJOR enhances the standard AspectJ weaver, without requiring any modification of neither the AspectJ language, nor the AspectJ compiler or weaver. For this, we adapted the existing AspectJ weaver as a FERRARI UDI. In this section we describe different transformations made during the weaving process and their relation to FERRARI features, which are fundamental to support aspect weaving with full coverage, and then we describe the implementation of the AJ-UDI and discuss its limitations.

### 4.1. Aspect Weaving

In the aspect class, advice are compiled into methods. The weaver inserts code to invoke these advice methods in the woven class. Aspect classes are used by the AJ-UDI to weave both application classes and the Java class library. Since advice methods are invoked by instrumented code, aspect classes are UDI-runtime-classes. Hence, the advice methods must not execute any instrumented code.

The execution of advice bodies implies the creation of an aspect instance. By default, there is a singleton aspect instance that is accessed by the woven classes through the static method **`aspectOf()`** that is generated by the aspect compiler in the aspect class. The AJ-UDI must ensure that the advice methods and the method **`aspectOf()`** do not execute any instrumented code in order to prevent infinite recursions. To temporarily bypass the execution of instrumented code, the developer may use the DIB API or alternatively use an automated tool (called **`AspectTransformer`**), which applies the code pattern shown in Figure 4 to every constructor, static initializer, advice method, and generated method in the aspect class.

Each kind of join point has its own corresponding bytecode representation (e.g., method execution – entire code segment of method body; field get – getfield or getstatic; etc.). Thus, the weaver analyzes and modifies the bytecode (1) to retrieve the aspect instance (call to **`aspectOf()`** in the aspect class) and (2) to invoke the advice methods, before, after, or around the corresponding bytecode representation of the join point.

If an advice makes use of AspectJ's reflective API to access static information about the join point (e.g., through the **`JoinPoint.StaticPart`** interface or when using the **`thisJoinPointStaticPart`** pseudo-variable), the weaver adds static fields in the woven class to store the corresponding references and modifies the static initializer. As described in Section 2, FERRARI offers support for UDIs to handle added static fields through extra classes. The AJ-UDI uses this feature for weaving the Java class library.

All the previously described weaving mechanism and code transformations, called dynamic-crosscutting, are not controlled by the aspect code, but are specific to the implementation of the weaver. AspectJ also supports static-crosscutting that enables structural transformations of types (classes, interfaces, and aspects) directly within the aspect description. The inter-type declaration mechanism allows adding static or instance fields, methods, and also modifying the class hierarchy with some restrictions. For weaving the Java class library, the AJ-UDI does not support static-crosscutting that modifies the class hierarchy.

### 4.2. Adapting the AspectJ Weaver

AspectJ supports compile-time and load-time weaving. The **`ajc`** compile-time tool is a compiler and bytecode weaver, i.e., it compiles and weaves applications and aspects from source code and can also weave aspects directly from bytecode. The **`ajc`** tool is based on an extension of the Eclipse Java compiler and an extension of Apache BCEL [2] for bytecode weaving. AspectJ supports two different approaches for load-time weaving: the first one uses a customized classloader and the second one is based on the **`java.lang.instrument`** API.

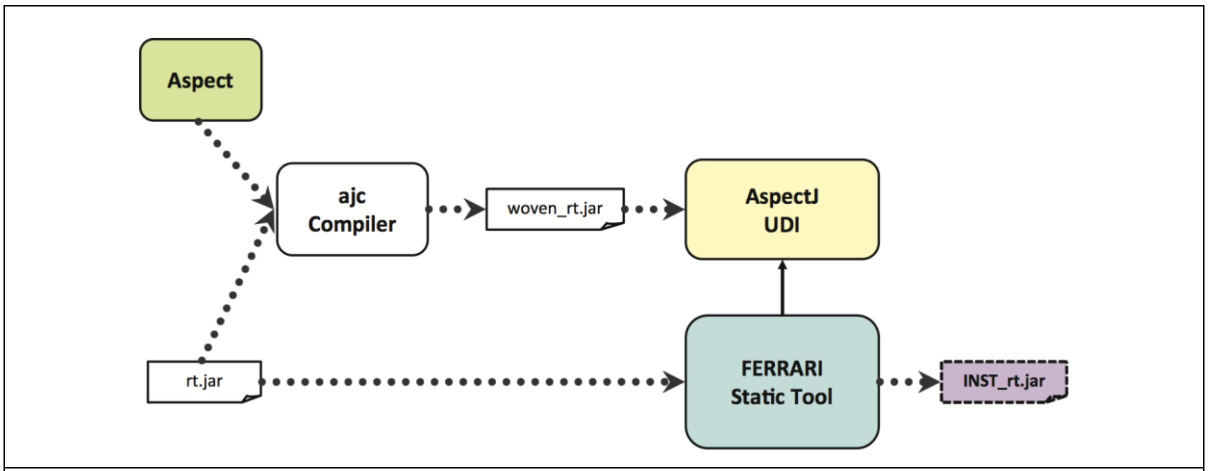

**Figure 8 –** Static instrumentation of the Java class library (**`rt.jar`**) with the AJ-UDI: First, **`ajc`** weaves the aspect into the Java class library (**`woven_rt.jar`**). Afterwards, FERRARI inserts the bypasses in the woven Java class library (**`INST_rt.jar`**).

The weaving of classes of the Java class library follows two different schemes. Here, we describe the base instrumentation one, which is based on two-phases scheme relying on static instrumentation. The second extended instrumentation scheme is described in Section 5.

The base instrumentation of the Java class library follows a two-phases scheme: first we use the **`ajc`** bytecode weaver to weave the aspect into the Java class library (see Figure 8 on the left side), and in the second phase, the AJ-UDI uses the woven Java class library such that FERRARI can insert the bypasses (see Figure 8 on the right side). Note that in the first phase, even though **`ajc`** weaves aspects into the Java class library, the resulting code is unusable as the standard aspect weaver does not handle the issues related to bootstrapping and to infinite recursions described in Section 3.2.

During the second phase, since no method signatures are modified by the weaver, the AJ-UDI performs a rather simple sequence: For each class to instrument, it reads the already woven version (instead of invoking the weaver) and compares it with the original class. It determines which methods were modified by the weaver, and checks if static fields were added (e.g., static fields related to AspectJ's reflective API described before). The AJ-UDI moves added static fields and the corresponding parts of the static initializers into extra classes. This information is provided to FERRARI. The resulting **`INSTR_rt.jar`** in Figure 8 has the woven classes with the bypasses together with the extra classes. Thanks to the JVM's lazy class initialization (see JVM Specification, Second Edition, Section 5.5) [17], the extra classes will be initialized after bootstrapping, because they are only used by UDI-inserted code, which is guaranteed not to execute during bootstrapping. Since FERRARI does not insert bypasses in the static initializers of extra classes, the UDI- inserted static fields in the extra classes will be properly initialized.

AspectJ provides a **`WeavingAdaptor`** class allowing third party applications to interact with the weaver. The adaptor receives a class as a byte array and returns the woven class also as a byte array. The AJ-UDI uses the **`WeavingAdaptor`** for dynamic instrumentation. FERRARI's runtime instrumentation agent plays a similar role as AspectJ's load-time mechanisms. Once a class is woven, the AJ-UDI determines which methods were modified by the weaver, and tells FERRARI to generate bypass code to enable callbacks from advice bodies into application code.

Figure 9 illustrates the execution of a dynamically woven application running on top of a woven Java class library. In this example we assume that the same aspect was woven in the application and in the Java class library (which is not necessarily always the case, as different aspects can be woven into the Java class library and the application). Both the woven Java class library and application code (**`INST_rt.jar`** and **`INST_App`**) invoke the advice in the aspect class, i.e., the UDI-runtime-class, which uses AspectJ's runtime system for full dynamic aspect support. Thus, FERRARI and the AJ-UDI provide advanced support for weaving the Java class library which is complementary to the features offered by existing AspectJ compilers and weavers.

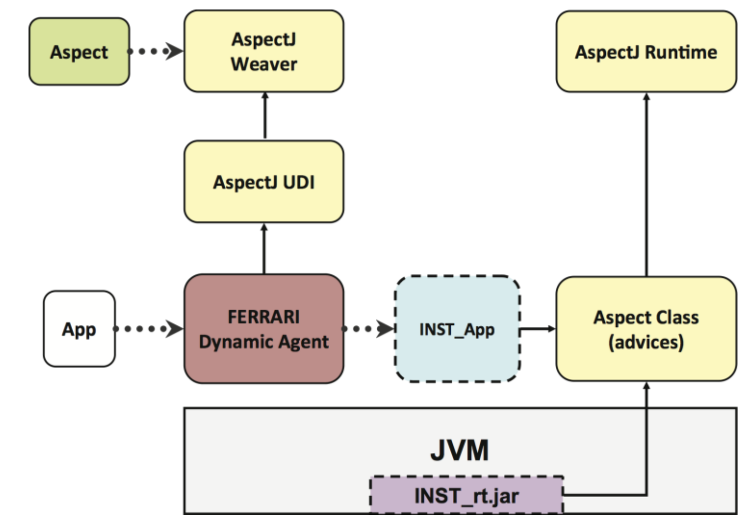

**Figure 9 –** Dynamic instrumentation with the AJ-UDI

In the next Section we describe the extended instrumentation scheme, allowing single-phase weaving of the Java class library. We detail the rationale behind this instrumentation scheme and how it allows to better isolate the instrumentation process from the execution of the instrumented code.

## 5. EXTENDED INSTRUMENTATION SUPPORT

FERRARI leverages the standard **`java.lang.instrument`** API and therefore the instrumentation of application classes is straightforward (application code is instrumented at load-time). However, for instrumenting classes of the Java class library to support full coverage, the use of the **`java.lang.instrumentation`** API introduces some difficulties. FERRARI solves the issues of bootstrapping with an instrumented version, by performing static instrumentation of the Java class library. That is, all classes of the Java class library are instrumented beforehand, and passed to the bootclasspath of the JVM. The static instrumentation notably adds the logic for bypassing instrumented code during bootstrapping using the Bootstrap Inserted-code Bypass (BIB) global flag.

Tanter introduced the notion of execution levels as a means to structure aspect-oriented programs so as to prevent infinite regressions and unwanted interference between aspects [25]. In [20], Polymorphic Bytecode Instrumentation (PBI) applies underlying instrumentation techniques similar to those used in FERRARI to support full coverage, in order enable the execution of different levels of instrumentations. The logic to switch between different execution levels can be seen as a generalization of the bypass mechanism of FERRARI. The application of PBI to aspect weaving also leverages the **`java.lang.instrument`** API, and therefore also uses static instrumentation to weave the Java class library as the two-phases instrumentation scheme described previously.

In general, the main reason for the static pre-processing step is because when using the **`java.lang.instrument`** API the instrumentation starts when the JVM has already bootstrapped, making it not possible to instrument the Java class library classes at load-time. Both FERRARI and PBI provide a mechanism to execute the original code during bootstrapping. However, in some cases, static instrumentation for aspect weaving can be cumbersome, because any modification on aspects requires to statically weaving the whole Java class library, which may also take a high amount of time and is tedious for the aspect developer. Some alternative approaches to avoid static instrumentation are based on (a) a customized JVM, (b) hotswapping techniques, and (c) native code. In the following, let us discuss their advantages and limitations.

Using a customized JVM gives high flexibility to integrate weaving support directly inside the JVM. Such an approach is used by Steamloom [21] and PROSE [13]. Even though this approach gives full control to the weaver developer and often introduces low weaving performance overhead compared to other approaches, modifying the JVM is rather tedious, error-prone, and limits the applicability of state-of-the-art JVM and instrumentation techniques (it is necessary to have access to the source code of the JVM and constantly update the modifications for every new release).

The second approach is based on hotswapping techniques [22], which in Java enables the redefinition of previously loaded classes. Hotswapping is supported by the JVM Tool Interface (JVMTI) [1], and by the **`java.lang.instrument`** API. While the JVMTI requires the use of native code, the **`java.lang.instrument`** API can be accessed in pure Java. The later solution has been applied in previous work on dynamic AOP with the HotWave weaver [29]. Unfortunately, current Java's hotswapping mechanism imposes several constraints: only method bodies may be modified; fields or methods cannot be added, removed, or renamed; method signatures or the class hierarchy cannot be changed. These restrictions limit the applicability of hotswapping to support all the transformations needed for aspect weaving. In addition, complex code transformations of the woven code are required.

The third approach to avoid static instrumentation of the Java class library is based on native code through JVMTI, namely load-time weaving using JVMTI. For this, JVMTI allows capturing classloading events and a user-defined JVMTI native agent receives the class representation as a byte array for instrumentation. Once the instrumentation is performed, the modified byte array is returned to be finally linked by the JVM. JVMTI enables the instrumentation itself to be implemented in native code or Java code. However, using native code for bytecode instrumentation is complex, tedious and error-prone. Thus, using Java code for the instrumentation is better suited and can be implemented by invoking Java code from native code using the Java Native Interface (JNI).

Among the three approaches to avoid static instrumentation, the first two (modified JVM and hotswapping) impose too many restrictions to be effectively applied to the underlying instrumentation mechanism of MAJOR. Considering a trade-off between portability and flexibility, the third approach based on native code, is therefore a good compromise. For aspect weaving with MAJOR, the use of native code in the underlying instrumentation layer avoids the use of a two-phases, static-based instrumentation for full coverage support. The main idea is therefore, that the JVMTI agent captures all classes that are loaded (including those of the Java class library), weaves the aspect on each class, and adds the logic to bypass instrumented code during bootstrap and to avoid infinite recursions. The instrumentation is therefore achieved at load-time for all classes uniformly. In contrast to the use of a **`java.lang.instrument`** agent to perform load-time the instrumentation (as the load-time weaving support in MAJOR for application code), this solution based on native code uses a minimal **`java.lang.instrument`** agent, which only signals the end of the bootstrap so as to start executing instrumented code in the Java class library classes.

Concerning the instrumentation itself, so far, we assumed that the instrumentation is performed in the same JVM where the instrumented code is executed. However, because the aspect weaver is itself implemented in Java, the use of JVMTI native code for instrumentation results in a circular dependency and therefore makes the approach not possible to happen on the same JVM. This is because the instrumentation of the very first class captured by the JVMTI agent (i.e. **`java.lang.Object`**) requires already several classes to be loaded (all those used by the weaver, including those from the Java class library, notably **`java.lang.Object`**). In addition, the loading of classes used by the instrumentation framework could introduce some perturbations on the loading of classes used by the application code. To solve these issues, the instrumentation is performed in a separate JVM. Thus, the JVM where the woven code is executed is completely independent from the JVM where the weaving happens. No perturbations are therefore possible induced by the loading of classes used for instrumentation.
The communication between the JVM where the woven code is executed (*client JVM*) and where the instrumentation happens (*server JVM*) is performed using an Inter-Process Communication (IPC) mechanism. We consider two different IPCs for this communication: *sockets* and *shared memory*. Communication through sockets has the advantage that the server side (server JVM) can be implemented in pure Java, because of the standard supports of sockets in the JDK. Additionally, socket communication enables the server JVM to be hosted on a different computer. This can be useful to reduce instrumentation time, for example, by hosting the instrumentation server JVM in a more powerful computer than the one running the client JVM. In contrast, the shared memory communication is faster than sockets, but requires the use of native code on both client and server JVM, and both JVMs must be hosted on the same computer.

Figure 10 illustrates the communication mechanism for instrumentation using socket IPC. On the client JVM, the native agent captures every classloading events using a JVMTI callback hook. The agent receives information about the captured class, which includes among others, the class as a byte array, the size of the class, its name, and the classloader reference. Because classloading can happen concurrently, the agent uses a thread-safe linked data structure to store connection specific information, including socket file descriptors to communicate with the instrumentation server JVM. If no socket is available on the structure, a new one is created, marked as unavailable, and used to send the class byte array to the server. Once the instrumentation is done and the woven class is returned using the same socket connection, the socket file descriptor is marked as available on the linked data structure. This allows reusing sockets connections and performing parallel transfers as much as possible.

The server JVM runs a multithreaded Java server, which uses a thread pool to run the instrumentations. The server uses the UDI interface to invoke the weaver (AJ-UDI) of MAJOR. All the logic to run instrumented code and handle bootstrapping issues is added to the woven code by the server. The low-level communication protocol between the client and server is straightforward. A packet header (encoded as 8 bytes) contains the size of the class name and the length of the class contained in the packet data. The values stored in the header are used to read the data fields containing the class name and the class byte array. Once instrumented, the same packet structure is used to send the woven class back. The new class length is used to read the new class from the socket, and is passed to the JVM for linking. Any exception during the instrumentation is reported to the client by setting a class length to zero. The exception string message is passed in the class name field of the packet. The client JVMTI agent reports the error to the user, and terminates the JVM.

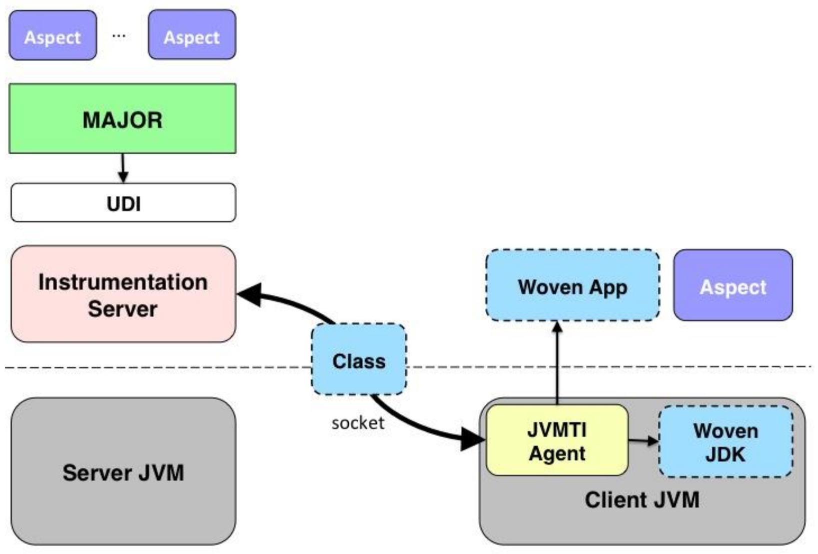

**Figure 10 –** Instrumentation in a separate JVM using socket IPC.

In the case of extra classes generated during instrumentation on the server JVM, the JVMTI agent receives the extra classes and puts then on a jar file, and appends it to the bootclasspath search path (with the **`AddToBootstrapClassLoaderSearch`** JVMTI function). This is because the class loading hook mechanism only allows a single instrumented class as result of the instrumentation.

UDI-runtime-classes, as is the case in the base instrumentation scheme, are excluded from normal instrumentation. Those classes are actually loaded and executed on the client JVM, and therefore are captured for instrumentation by the JVMTI agent. Note that the aspect class must be available on both the server and the client JVM. In the server JVM, because it is used by the AspectJ to perform the weaving process (the aspect class contains metadata controlling the weaving process). In the client JVM, because the aspect class holds advice code, and therefore must be instantiated when executing woven code.

Figure 11 shows the communication mechanism based on shared memory. This solution uses two communication channels: a socket to exchange control data, and the shared memory for the actual data exchange. The socket communication is used to send information about the shared memory segment to be used for the communication and for error message communication. The shared memory is used to exchange the byte array of class to be instrumented and the instrumented one. Because there is no standard support for shared memory in Java, the instrumentation server uses Java Native Interface (JNI) to call shared memory read and write operations. That is, when the instrumentation request arrives to the server (through a Java socket) the information sent through it is used to read the actual data using a native method. Similarly, once the AJ-UDI has finished weaving a class, the information about the instrumented class (notably the new size) is sent via the socket, and the actual instrumented class is written to the shared memory through a native method.

Thanks to the presented extended instrumentation approach, the weaving process is completely isolated from the execution of the woven code and the instrumentation process is uniformly performed at load-time for both the application code and the Java class library. This requires the use of a tiny native code layer, compared to the pure Java static instrumentation approach. Even though the extended instrumentation approach uses two JVMs, the setup of the instrumentation is simplified and very similar to the base instrumentation approach with a single JVM. The user has therefore the choice of using the more adapted scheme, according to his/her needs.

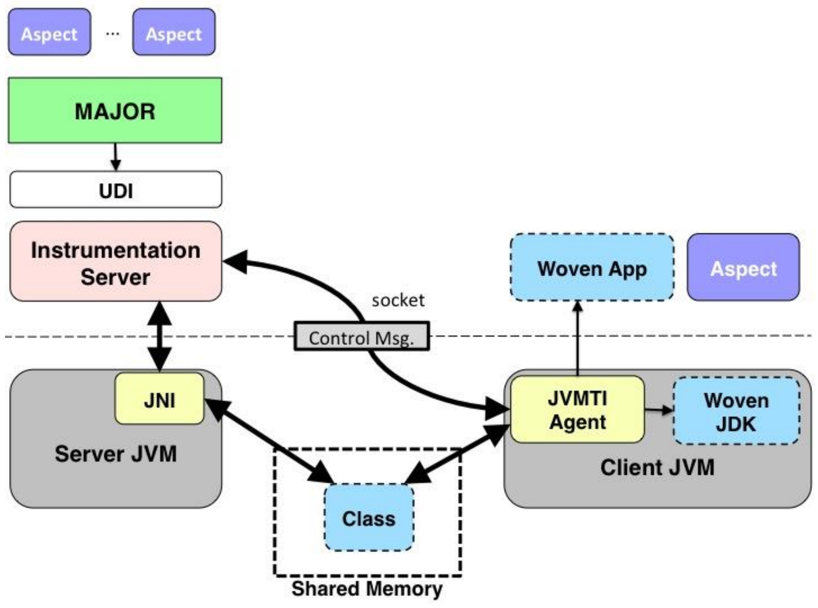

**Figure 11 –** Instrumentation in a separate JVM using shared memory IPC.

## 6. RELATED WORK

The “Twin Class Hierarchy” (TCH) [23] claims to support user-defined instrumentation of the standard Java class library. TCH replicates the full hierarchy of the instrumented Java class library in a separate package that coexists with the original one. However, this technique has the disadvantage that applications need to be instrumented to explicitly refer to a desired version of the Java class library (original or instrumented). In addition, as pointed out in [24], the use of replicated classes limits the applicability of instrumentation in the presence of native code; e.g., call-backs from native code may not reach the instrumented code. Consequently, TCH fails to transparently instrument the complete Java class library and is therefore not suited for aspect weaving with full coverage.

Unlike TCH, our approach does not use class replication techniques, but rather code duplication within the method bodies. We enable aspect weaving with full coverage independently of the presence of native methods. In addition, our approach ensures transparency for the application, which need not be aware whether the Java class library is instrumented or not.

PROSE allows runtime weaving of aspects [13], that is, aspects can be woven during the application execution. To this end, PROSE uses code hotswapping techniques, bytecode instrumentation, and an extension of the Jikes RVM [30] to support code replacement while the application is running. Even though hotswapping removes the problem of bootstrapping with an instrumented Java class library (hotswapping is triggered after bootstrapping), it also imposes strong restrictions on the possible transformations (fields and methods cannot be added or removed), thus complicating the applicability in support of aspect weaving with full coverage. In addition, PROSE does not support standard AspectJ, but defines aspects and transformations as regular Java classes.

The work on MAJOR has largely inspired our work on HotWave [29], a comprehensive runtime aspect weaver. HotWave relies on hotswapping through class redefinition provided by the **`java.lang.instrument`** API. HotWave relies on standard AspectJ and avoids the problem of bootstrapping with an instrumented Java class library. Similar to the single-phase weaving presented in this article, HotWave does not require static instrumentation.

Hotwave2 [31,28] is an enhancement of HotWave based on a customized, production quality JVM, the Dynamic Code Evolution VM [32], which removes most of the hotswapping restrictions of prevailing JVMs.

JFluid [26] uses hotswapping to dynamically turn on and off profiling code to collect dynamic metrics for selected code regions. JFluid has been integrated into the state-of-the-art NetBeans Profiler [27]. The architecture of JFluid is similar to our single-phase instrumentation based on Inter-Process Communication (IPC) communication using sockets and shared memory using a separate JVM for instrumentation. JFluid however, uses a fixed instrumentation scheme, whereas our approach enables easy implementation of aspect-based profiling tools. In addition, JFluid focuses on profiling of application code, whereas we provide aspect weaving with full coverage.

Steamloom [21] provides aspect support within the JVM, which may ease calling context reification thanks to the direct access to JVM internals. Steamloom is an extension of the Jikes RVM [30] supporting efficient aspect execution and runtime aspect weaving. This approach trades portability for performance, since the aspect support is integrated in the JVM.

In order enable different execution levels [25], Polymorphic Bytecode Instrumentation (PBI) [20] applies underlying instrumentation techniques similar to those used in FERRARI to support full coverage. The application of PBI to aspect weaving uses static instrumentation to weave the Java class library similar to the two-phases instrumentation scheme presented in this paper. The single-phase instrumentation scheme can be however easily adapted to PBI.

## 7. CONCLUSION

In this article we presented techniques and tools that enable aspect weaving with full coverage. Our approach relies on a generic framework that enhances user-defined instrumentations with complete method coverage. We adapted the popular AspectJ weaver to our framework MAJOR without changing any AspectJ sources. We succeeded in removing a serious limitation from AspectJ, its inability to weave aspects into the Java class library. Our approach makes aspect-based profiling and debugging techniques practical, since it ensures full coverage. Moreover, it makes AOP available to the developers of the Java class library. We presented two approaches to deal with the issues of bootstrapping with instrumented Java class library, a first one based on a two-phases instrumentation scheme using a static tool, and a second one based on a single-phase instrumentation using a tiny native code layer. With both approaches available, MAJOR gives flexibility to the aspect-based tool developer to choose the appropriate scheme according to his/her requirements.

## 8. ACKNOWLEDGEMENTS

We want to thank Aibek Sarimbekov for his help on the implementation of the single-phase instrumentation scheme. Part of this work has been supported by the Swiss National Science Foundation.

## 9. REFERENCES

[1] Oracle Corporation. “JVM Tool Interface (JVMTI) version 1.2.1.” Internet: http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html , 2007.

[2] The Apache Commons Project. “The Byte Code Engineering Library (BCEL).” Internet: http://commons.apache.org/bcel/.

[3] OW2 Consortium. “ASM – A Java bytecode engineering library.” Internet: http://asm.ow2.org/.

[4] Jboss. “Javassist Project.” Internet: at http://www.jboss.org/javassist.

[5] M. Denker et al. “Runtime bytecode transformation for Smalltalk.” *Journal of Computer Languages, Systems and Structures*, vol. 32, no. 2-3, pp. 125–139, July 2006.

[6] G. Kiczales. et al. “Aspect-oriented programming.” In M. Akşit and S.Matsuoka, editors ECOOP’97: Proceedings of the 11th European Conference on Object-Oriented Programming, volume 1241 of Lecture Notes in Computer Science, pp. 220–242, Jyväkylä, Finnland, June 1997. Springer-Verlag

[7] G. Kiczales. et al. “An overview of AspectJ.” In J. L. Knudsen, editor, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001), volume 2072 of Lecture Notes in Computer Science, pp. 327–353, 2001.

[8] E. Hilsdale and J. Hugunin. “Advice weaving in AspectJ.” In AOSD ’04: Proceedings of the 3rd International Conference on Aspect- Oriented Software Development, pp. 26–35, New York, NY, USA, 2004. ACM.

[9] D. J. Pearce et al. “Profiling with AspectJ.” *Software: Practice and Experience*, vol. 37, no. 7, pp. 747–777, June 2007.

[10] G. Ammons et al. “Exploiting hardware performance counters with flow and context sensitive profiling.” In PLDI ’97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pp. 85–96. ACM Press, 1997.

[11] S. Artzi et al. “ReCrash: Making Software Failures Reproducible by Preserving Object States.” In J. Vitek, editor, ECOOP ’08: Proceedings of the 22th European Conference on Object-Oriented Programming, volume 5142 of Lecture Notes in Computer Science, pp. 542–565, Paphos, Cyprus, 2008. Springer-Verlag.

[12] W. Binder et al. “Advanced Java Bytecode Instrumentation.” In PPPJ’07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp. 135–144, New York, NY, USA, 2007. ACM Press.

[13] A. Popovici et al. “Just-in-time aspects: efficient dynamic weaving for Java.” In AOSD ’03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 100–109, New York, NY, USA, 2003. ACM Press.

[14] A. Villazón et al. “Aspect Weaving in Standard Java Class Libraries.” In PPPJ ’08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp. 159–167, New York, NY, USA, Sept. 2008. ACM.

[15] A. Villazón et al. “Comprehensive Aspect Weaving for Java,” in *Science of Computer Programming*, vol. 76, no. 11, pp. 1015–1036, Elsevier North-Holland, Inc. Nov. 2011. Amsterdam, The Netherlands.

[16] R. Valleée-Rai et al. “Optimizing Java bytecode using the Soot framework: Is it feasible?” In *Compiler Construction*, 9th International Conference (CC 2000), pp. 18–34, 2000

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[18] G. Xu and A. Rountev. “Precise memory leak detection for Java software using container profiling.” In ICSE ’08: Proceedings of the 30th international conference on Software engineering, pp. 151–160, New York, NY, USA, 2008. ACM.

[19] D. L. Heine and M. S. Lam. “Static detection of leaks in polymorphic containers.” In ICSE ’06: Proceedings of the 28th international conference on Software engineering, pp. 252–261, New York, NY, USA, 2006. ACM.

[20] P. Moret et al. “Polymorphic Bytecode Instrumentation.” In AOSD ’11: Proceedings of the 10nd International Conference on Aspect-Oriented Software Development, pp. 129–140, New York, NY, USA, 2011. ACM Press

[21] M. Haupt et al. “An execution layer for aspect-oriented programming languages.” In VEE ’05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, pp. 142–152, New York, NY, USA, 2005. ACM.

[22] M. Dmitriev. “Profiling Java applications using code hotswapping and dynamic call graph revelation.” In WOSP ’04: Proceedings of the Fourth International Workshop on Software and Performance, pp. 139–150. ACM Press, 2004.

[23] M. Factor et al. “Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach.” In OOPSLA ’04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 288–300, New York, NY, USA, 2004. ACM.

[24] E. Tilevich and Y. Smaragdakis. “Transparent program transformations in the presence of opaque code.” In GPCE ’06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, pp. 89–94, New York, NY, USA, 2006. ACM

[25] E. Tanter. “Execution levels for aspect-oriented programming.” In Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010), pp. 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[26] M. Dmitriev. “Profiling Java applications using code hotswapping and dynamic call graph revelation.” In WOSP ’04: Proceedings of the Fourth International Workshop on Software and Performance, pp. 139–150. ACM Press, 2004

[27] NetBeans, “The NetBeans Profiler Project.” Internet: http://profiler.netbeans.org/.

[28] T. Würthinger et al. “Improving Aspect-Oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine.” In RAM-SE’10: Proceedings of the 7th ECOOP, Workshop on Reflection, AOP and Meta-Data for Software Evolution, Maribor, Slovenia, 2010

[29] A. Villazón et al. “Advanced Runtime Adaptation for Java.” In GPCE ’09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering, pp. 85-94. ACM, Oct. 2009.

[30] Jikes RVM. “Research Virtual Machine.” Internet: http://jikesrvm.org/.

[31] T. Würthinger et al. “Applications of enhanced dynamic code evolution for Java in GUI development and dynamic aspect-oriented programming.” In Proceedings of the 9th ACM SIGPLAN International Conference on Generative

Programming and Component Engineering (GPCE 2010), pp. 123–126, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[32] T. Würthinger et al. “Safe and Atomic Run-time Code Evolution and its Application to Dynamic AOP.” In OOPSLA ’11: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 825-844, New York, NY, USA, 2011. ACM Press.