Advanced Runtime Adaptation for Java

Alex Villazón Walter Binder Danilo Ansaloni Philippe Moret

Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
firstname.lastname@usi.ch

Abstract

Dynamic aspect-oriented programming (AOP) enables runtime adaptation of aspects, which is important for building sophisticated, aspect-based software engineering tools, such as adaptive profilers or debuggers that dynamically modify instrumentation code in response to user interactions. Today, many AOP frameworks for Java, notably AspectJ, focus on aspect weaving at compile-time or at load-time, and offer only limited support for aspect adaptation and reweaving at runtime. In this paper, we introduce HotWave, an AOP framework based on AspectJ for standard Java Virtual Machines (JVMs). HotWave supports dynamic (re)weaving of previously loaded classes, and it ensures that all classes loaded in a JVM can be (re)woven, including the classes of the standard Java class library. HotWave features a novel mechanism for inter-advice communication, enabling efficient data passing between advices that are woven into the same method. We explain HotWave's programming model and discuss our implementation techniques. As case study, we present an adaptive, aspect-based profiler that leverages HotWave's distinguishing features.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.3 [*Language Constructs and Features*]: Frameworks

General Terms Algorithms, Languages, Measurement

Keywords Dynamic aspect-oriented programming, runtime aspect adaptation and (re)weaving, bytecode instrumentation, code hotswapping, AspectJ, Java Virtual Machine

1. Introduction

Aspect-oriented programming (AOP) [18] enables the specification of cross-cutting concerns in applications, avoiding related code that is scattered throughout methods, classes, or components. Traditionally, AOP has been used for disposing of "design smells", such as needless repetition, and for improving maintainability of applications. AOP has also been successfully applied to the development of *software engineering tools*, such as profilers, debuggers, or testing tools [27, 4, 35], which in many cases can be specified as aspects[1] in a concise manner. Hence, in a sense, AOP can be regarded as a versatile approach for specifying certain *program transformations* at a high level, hiding low-level implementation details, such as bytecode manipulation, from the programmer.

Dynamic AOP allows aspects to be changed and code to be rewoven in a running system, thus enabling runtime adaptation of applications. Dynamic AOP has been used for adding persistence or caching at runtime [23], or for debugging and fixing bugs in a running server [10]. Regarding aspect-based software engineering tools that collect information on the dynamic behavior of applications, such as profilers, dynamic AOP enables tools where developers can refine the set of dynamic metrics of interest, and choose the components to be analyzed, while the target application is executing. Such features are essential for analyzing complex, long-running applications, where the comprehensive collection of dynamic metrics in the overall system would cause excessive overheads and reduce developers' productivity. In fact, state-of-the-art profilers, such as the NetBeans Profiler [22], rely on such dynamic adaptation, but currently these tools are implemented with low-level instrumentation techniques, which cause high development effort and costs, and hinder customization and extension.

While dynamic AOP is often supported in dynamic languages [14, 5], such as Lisp or Smalltalk, where code can be easily manipulated at runtime, it is more difficult and challenging to offer dynamic AOP for languages like Java. Several popular AOP frameworks for Java, such as AspectJ [17] or *abc* [3], do not support dynamic AOP, that is, aspects are woven statically at compile-time or at load-time, but cannot be (re)woven in a running system. While there are AOP frameworks for Java that explicitly support dynamic AOP, such as PROSE [24] or Steamloom [9], they often restrict the aspect language or limit portability by relying on a dedicated, modified Java Virtual Machine (JVM) or by using native code.

In this paper, we introduce HotWave (standing for HOTswap & reWeAVE), a new system offering dynamic AOP in Java. HotWave is unique in reconciling three important features: *dynamic AOP*, *compatibility*, and *complete method coverage*. HotWave is based on AspectJ and supports a wide range of standard AspectJ constructs for dynamic cross-cutting. HotWave is portable, implemented in pure Java, and is compatible with standard, state-of-the-art JVMs. In contrast to other AOP frameworks for Java, HotWave enables aspect (re)weaving with complete method coverage, that is, aspects can be woven into any method that has a bytecode representation, including methods in dynamically generated classes or in the standard Java class library.

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.

[1] Aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, field accesses, etc. *Advices* are executed *before*, *after*, or *around* the intercepted join points. Advices have access to contextual information of the join points.

These distinguishing features make HotWave an ideal framework for the rapid development of sophisticated software engineering tools.

HotWave features a new mechanism, *inter-advice communication*, for efficiently passing data between advices that are woven into the same method. With the aid of annotations, the aspect programmer can declare *invocation-local* variables, which correspond to local variables with the scope of a woven method. To implement inter-advice communication, HotWave selectively inlines advices that access invocation-local variables. HotWave's inter-advice communication model is complementary to AspectJ constructs and enables optimizations in aspect-based software engineering tools.

The original, scientific contributions of this paper are threefold:

1. We present HotWave, a new aspect weaver that supports dynamic AOP in Java and enables aspect adaptation in a running system. HotWave's programming model is based on AspectJ. Our implementation in pure Java relies on standard, unmodified AspectJ weaving tools and is compatible with standard JVMs. HotWave applies program transformations at the bytecode level to woven code in order to meet Java's constraints on runtime class redefinition.
2. We introduce a novel AOP mechanism, inter-advice communication, for efficiently passing data between advices in local variables.
3. As case study, we present an adaptive profiler that relies on HotWave's distinguishing features. Our case study shows that sophisticated software engineering tools can be concisely specified as HotWave aspects, which can be easily extended.

This paper is structured as follows: Section 2 discusses background and related work on dynamic AOP. Section 3 presents HotWave's programming model, summarizing the supported features and discussing its limitations. Section 4 explains HotWave's inter-advice communication mechanism and illustrates its benefits with two examples. Section 5 details HotWave's architecture and implementation. Section 6 presents our case study, an adaptive profiler implemented as an aspect for HotWave. Finally, Section 7 concludes this paper.

2. Background and Related Work

In this section we give an overview of code hotswapping techniques, which are used in dynamic AOP frameworks for Java. We discuss existing frameworks, their strengths, and their limitations. We also review some techniques developed in prior work that are relevant for this paper.

2.1 Code Hotswapping

Code hotswapping enables dynamic evolution of a running application. Hotswapping is supported by dynamic languages such as Erlang, Smalltalk, and Lisp. In Java, hotswapping enables the redefinition of previously loaded classes and is supported by the JVM Tool Interface (JVMTI) [31], which is part of the Java Portable Debugging Architecture (JPDA) [30], 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 combination of dynamic bytecode instrumentation with code hotswapping enables advanced debugging, profiling, and monitoring capabilities. For example, JFluid [11] 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 [22].

Currently, 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.

2.2 Dynamic AOP

Dynamic AOP enables runtime adaptation of a running application without stopping or restarting it. There are three different approaches to dynamic AOP.

1. *Pre-runtime instrumentation* inserts small pieces of code (hooks) at every code location which may become a join point later. The insertion can be based on pre-processing [26, 16], on load-time instrumentation [26], or on just-in-time compilation [28].
2. *Runtime event monitoring* [29] uses low-level JVM support to capture events such as method entry, method exit, and field access.
3. *Runtime weaving* [14, 10, 24] dynamically weaves aspects into already executing code.

The latter approach is the most challenging and can be implemented either with standard hotswapping support or with a customized JVM.

PROSE [29, 28, 24] is an adaptive middleware platform offering dynamic AOP. PROSE has evolved in three versions, which make use of the three aforementioned approaches to dynamic AOP. The last version of PROSE [24] uses bytecode instrumentation techniques to support code replacement while the application is running. PROSE inlines a special kind of advice for method replacement. PROSE defines aspects and transformations as regular Java classes, that is, the aspect language of PROSE is Java. While this approach allows using standard Java compilers, it results in a verbose aspect programming model. PROSE has been implemented with a modified Jikes RVM [1] and with the JPDA hotswapping mechanism, which requires native code. PROSE does not support static join points[2], has a limited reflective interface, lacks support for control flow pointcuts, and provides limited access to context information within advices.

Steamloom [9] is one of the few frameworks to provide AOP support at the JVM level, which results in efficient runtime weaving. It enables runtime code replacement through an extension of the Jikes RVM. Steamloom uses its own aspect language and provides a parser to support AspectJ-like pointcuts. It however does not support static join points.

Wool [10] combines execution monitoring with hotswapping. It sets breakpoints at all join points identified by a pointcut, so as to weave calls to advice code on the replacement methods once an activated join point reaches a break point. JPDA is used for monitoring and Javassist [15] is used for instrumentation. Wool uses Java as aspect language and supports a limited set of join points and advices compared to AspectJ. Static join points, around advices, and control flow pointcuts are not supported.

JBossAOP [16] supports compile-time, load-time, and runtime weaving. JBossAOP uses the hotswapping mechanism of the `java.lang.instrument` API for runtime weaving. JBossAOP uses Java as aspect language together with annotations and XML-based pointcut definitions. The JBossAOP weaver may insert auxiliary fields and methods, but given the JVM constraints for hotswapping, JBossAOP forces a preparation step (at compile-time or at load-time) so as to know in advance the join points and add the cor-

[2] In this paper, the term *static join point* denotes the static part of a join point. In AspectJ, advices may access static join points through the `thisJoinPointStaticPart` pseudo-variable.

responding auxiliary fields. This approach restricts runtime weaving only to aspects with exactly the same pointcuts.

AspectWerkz [33] allows to deploy and undeploy aspects at runtime using hotswapping. Similar to JBossAOP, AspectWerkz forces a preparation step at load-time to add auxiliary fields to support different pointcuts. AspectWerkz uses Java with annotations and XML as aspect language.

Runtime weaving in JAsCo [32] relies on the hotswapping support of JPDA or of the `java.lang.instrument` API. If JPDA is used, JAsCo requires preliminary load-time insertion of hooks for aspect weaving. JAsCo uses its own aspect language, does not support static join points, and therefore heavily relies on reflection to gather join point information.

All existing dynamic AOP frameworks define their own aspect language, often supporting only a subset of the AspectJ constructs, and implement their own code weaver. HotWave takes the opposite approach by using the well established AspectJ language, together with existing compiler and weaver tools (without any modification), and transforms the resulting woven code in order to conform to the restrictions imposed by Java's hotswapping mechanism. Hence, HotWave benefits from state-of-the-art developments in AspectJ and does not require AspectJ developers to learn another language or to modify existing aspect code.

2.3 Prior Work by the Authors

A distinguishing feature of HotWave is its ability to weave aspects into any method that has a bytecode representation, including methods in the Java class library. To this end, HotWave relies on similar techniques as FERRARI [7], a generic bytecode instrumentation framework that guarantees complete method coverage. In contrast to FERRARI, which requires static instrumentation of the Java class library, HotWave relies on dynamic instrumentation at runtime and on code hotswapping.

MAJOR[3] [34, 35] offers aspect weaving with complete method coverage; it is based on AspectJ and on FERRARI. In contrast to HotWave, MAJOR supports neither dynamic AOP, nor efficient inter-advice communication.

BMW [6] is a profiler generator that provides only a limited set of low-level pointcuts. BMW offers neither complete method coverage, nor supports hotswapping.

In [8, 21], we addressed platform-independent calling context profiling using a hard-coded, low-level instrumentation that was difficult to extend. Thanks to AOP and to our new inter-advice communication, it is possible to concisely express profilers as aspects in just a few lines of code.

3. HotWave Programming Model

In this section, we describe the HotWave aspect programming model. We summarize the supported features, explain dynamic weaving, and discuss current limitations.

3.1 Supported Features

- **AspectJ compatibility:** HotWave aspects are programmed in AspectJ, and the framework leverages the unmodified AspectJ compiler and weaver.
- **Runtime weaving**: HotWave supports aspect (re)weaving at runtime without interrupting an executing application. HotWave relies on Java's hotswapping mechanism to weave a new aspect into previously loaded classes. As an optimization, selected classes can be excluded from reweaving and hotswapping, if it is statically known to the aspect programmer that reweaving with the new aspect will not change these classes.

[3] `http://www.inf.unisi.ch/projects/ferrari/`

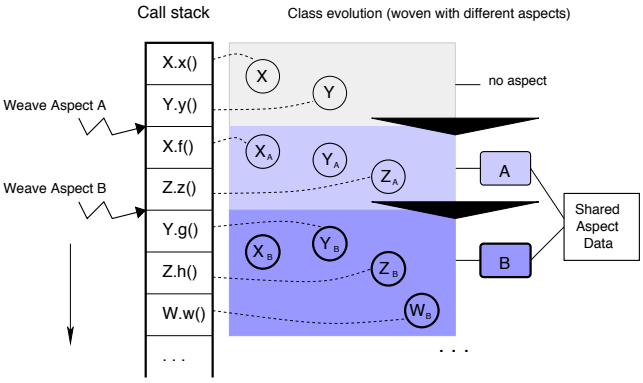

Figure 1. Example call stack referring to different aspects that are woven at runtime

Newly loaded classes are always woven with the most recent aspect version at load-time. HotWave also efficiently supports unweaving, that is, hotswapping all woven classes with the original classes.

- **Singleton aspects**: HotWave supports singleton aspects.
- **Pointcut designators:** HotWave supports the standard AspectJ pointcut designators (`execution`, `call`, `get`, `set`, `handler`, `cflow`, `if`, `within`, `this`, `target`, `args`, etc.).
- **Join points**: HotWave supports AspectJ's reflective API, which includes full access to static and dynamic join point information within advices (`thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`). Join points are very important for building aspect-based tools that require access to context information.
- **Advices:** HotWave supports `before` and `after` advices. The `around` advice is not directly supported, but HotWave introduces a new mechanism, inter-advice communication, which allows emulating a common use of `around` advices with a combination of `before` and `after` advices.
- **Inter-advice communication**: HotWave allows to efficiently communicate data between advices. HotWave introduces the notion of *invocation-local* variables (in analogy to thread-local variables), which are local to a method invocation. Inter-advice communication allows advices to store/load any data in/from local variables within the scope of an invoked woven method.
- **Complete method coverage**: HotWave allows runtime weaving of all classes without any restriction, including the classes in the standard Java class library. Since the weaver itself uses eventually woven classes in the standard library, HotWave provides a mechanism to bypass inserted code in order to avoid infinite recursions and perturbations due to runtime weaving.

3.2 Dynamic Weaving and Hotswapping

HotWave offers two mechanisms to trigger aspect weaving.

1. An initial aspect can be specified to be woven after completion of JVM bootstrapping and before any application code is executed. The initial aspect is woven both into classes that have been previously loaded during bootstrapping (which will be hotswapped), as well as into subsequently loaded classes.
2. (Re)weaving can be triggered at any time while application code is already executing. The new aspect version is woven into previously loaded classes that are hotswapped, as well as into

all subsequently loaded classes at load-time. Each time a class is woven, the original class bytes (and not the resulting class bytes from an eventual previous weaving step) are the input for the weaving.

In contrast to other dynamic AOP frameworks, such as JBossAOP [16] and AspectWerkz [33], HotWave does not apply any special preparatory step at load-time, such as inserting hooks or extra fields to enable later (re)weaving at runtime.

The insertion of static fields is a common technique used by aspect weavers to store static join point information [13, 3]. HotWave applies a special transformation after weaving, moving inserted static fields into separate classes, in order to ensure compatibility of aspect weaving with Java's constraints on hotswapping. Thus, distinct aspects that are woven at runtime may have arbitrarily different pointcut definitions.

Figure 1 shows the evolution of classes woven with different aspects at runtime. It illustrates the call stack of an executing thread, which invokes methods in different classes. Assuming only the aforementioned second weaving mechanism is used, the weaving is triggered while the thread is already executing unmodified code (methods `X.x()` and `Y.y()` in Figure 1). When aspect A is woven into all classes, the previously loaded classes X and Y are woven and hotswapped, resulting in the woven classes X_A and Y_A. Subsequently loaded classes are woven at load-time (e.g., class Z is woven into Z_A). Method `X.f()` executes the woven code in X_A, because it is invoked after hotswapping. The callers of `X.f()` on the stack (`X.x()` and `Y.y()`) are not affected by the hotswapping. That is, when method `X.f()` completes and method `Y.y()` resumes execution, the unmodified code in class Y is executed again.

Figure 1 also illustrates reweaving with a different aspect B, yielding the rewoven classes X_B, Y_B, and Z_B. Subsequent calls to methods in these classes execute advices in B, whereas the callers on the stack before the reweaving was triggered will continue to execute advices in A. That is, reweaving does not instantaneously replace an aspect in a running system, but multiple aspect versions may coexist at runtime. For this reason, each aspect version must have a different name[4] (i.e., aspects are not hotswapped, only woven classes are hotswapped). Furthermore, it is essential that different versions of an aspect use a common data representation that is shared by the aspects, as illustrated in Figure 1 on the right.

In HotWave, only the most recent aspect version is considered for weaving. Hence, when class W is loaded in Figure 1, it is woven with aspect B, resulting in class W_B.

3.3 Limitations

Concerning the limitations of HotWave's programming model, HotWave does not support non-singleton aspect instances using `per*` clauses (e.g., per-object or per-control flow aspect association).

HotWave does not directly support `around` advices, since the AspectJ weaver may insert wrapper methods for these advices, which would violate the constraints of Java's hotswapping mechanism. Nonetheless, with the aid of HotWave's inter-advice communication feature, it is possible to emulate a common use of `around` advices.

Static cross-cutting (inter-type declarations) [17] enables explicit structural modifications, such as changes of the class hierarchy or insertions of new fields and methods. As any other dynamic AOP framework based on hotswapping, HotWave does not support static cross-cutting due to the restrictions imposed by Java's hotswapping mechanism, which only allows modifications of method bodies.

[4] This constraint can be easily addressed by appending a version number to the aspect name.

4. Inter-Advice Communication

In AspectJ, the `around` advice (in conjunction with a `proceed` statement) allows storing data in local variables before a join point, and accessing that data after the join point. Hence, one common use of the `around` advice can be regarded as communicating data produced in a `before` advice to an `after` advice, within the scope of a woven method. However, prevailing AOP frameworks do not provide any general support for efficiently passing data in local variables between arbitrary advices that are woven into the same method body. For instance, in AspectJ it is not possible to pass data in local variables from one "`before call`" advice to an "`after execution`" advice.

There are two common approaches to advice weaving: At each join point in a woven method, (a) an invocation to an advice is inserted, or (b) the advice body is inlined. For example, AspectJ always follows the first approach [13], *abc* applies some heuristics to decide whether inlining should be applied or not [3], and PROSE uses inlining for method replacement and advice invocation otherwise [24]. As illustrated in Figure 2, passing data in local variables from one advice to another is only possible if the advices are inlined in the same woven method. If an advice is invoked as a method, the local variables used by the advice method are not accessible after its completion.

HotWave offers a unique feature, *inter-advice communication*, for efficiently passing data in local variables between any advices. To this end, HotWave selectively inlines[5] advice methods that refer to specially marked static fields, and allocates local variables for these special fields. We call these fields *invocation-local* variables (in analogy to thread-local variables), since their scope is the execution of one invoked woven method.

We use the custom Java annotation `@InvocationLocal` to declare public static fields in an aspect as invocation-local variables. Within advices, invocation-local variables can be read and written as if they were static fields. `@InvocationLocal` is defined as follows:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.CLASS)
public @interface InvocationLocal {}
```

Aspects are first woven with the standard AspectJ weaver, as illustrated in Figure 3 on the left. Afterwards, program transformations inline those advices that access invocation-local variables and map the invocation-local variables to local variables, as shown in Figure 3 on the right. The local variables are initialized with the values that have been assigned to the static invocation-local fields by the aspect's static initializer. Since advices accessing invocation-local variables are inlined in woven methods, HotWave requires that these advices do no access/invoke any non-public fields/methods of the aspect.

In the following we give two examples for inter-advice communication.

Emulating around advices: The `around` advice surrounds join points. Figure 4(a) shows a common use of the `around` advice, involving pre- and post-processing of data by surrounding every method invocation. In the example, the elapsed wall clock time is logged for every method.

The AspectJ weaver implements the `around` advice by adding wrapper methods in woven classes [13], which would violate Java's hotswapping constraints.

[5] Inlining in HotWave is not aimed at eliminating the overhead of advice invocation (such optimizations are anyway provided by state-of-the-art just-in-time compilers), but it is a prerequisite for efficient inter-advice communication using local variables.

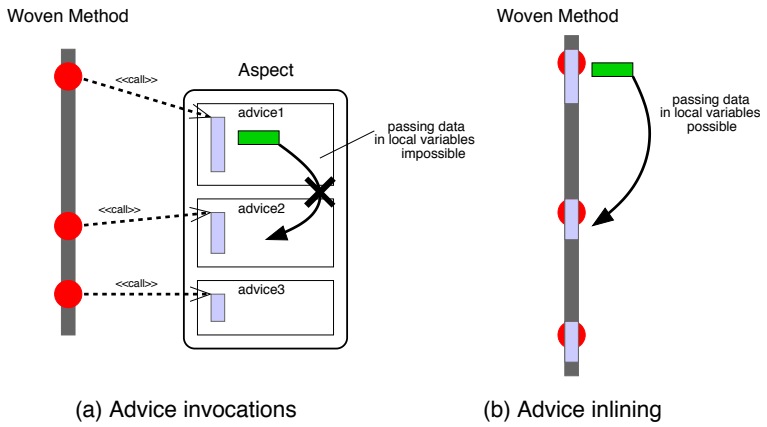

Figure 2. Advice weaving approaches

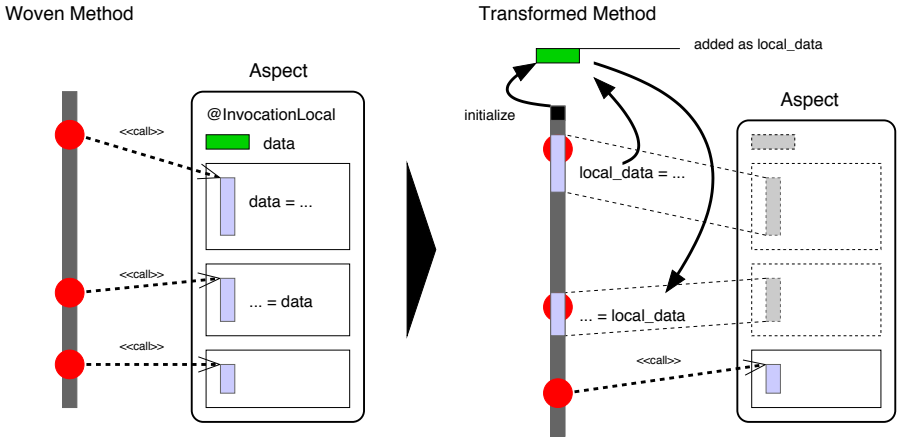

Figure 3. Transformations for inter-advice communication

Figure 4(b) shows a solution based on inter-advice communication that emulates an `around` advice with `before` and `after` advices. By declaring the `start` variable as `@InvocationLocal`, we ensure that the value stored in the `before` advice is accessible in the `after` advice. We use this mechanism to enable some basic `around` advice functionality for runtime weaving with HotWave. More sophisticated usage of the `around` advice, such as totally bypassing a join point, cannot be emulated with inter-advice communication.

Efficiently emulating cflow pointcuts: The `cflow` pointcut captures all join points in the control flow of a given join point. Former AspectJ weaver implementations used a thread-local shadow stack to verify whether a join point was in the corresponding control flow or not [20]. This approach is expensive because of the update of the shadow stack and the required dynamic checks. *abc* applies an optimization for the `cflow` pointcut by using a counter instead of a shadow stack [3]. The counter is incremented (respectively decremented) when the join point defining the control flow is entered (respectively left). A positive counter value means that a join point is in the corresponding control flow. The counter is stored in a thread-local variable and read only when it is first used in a given method.

Inter-advice communication allows to emulate a `cflow` pointcut and to apply the same optimization as in *abc*. Figure 5 shows an example of `cflow` emulation. We use two invocation-local variables: one to cache the counter and the other to read the thread-local counter only once. The equivalent AspectJ code is also shown. Note that *abc* applies this technique as a compiler optimization, which is not visible to the programmer, whereas our approach provides a simple, yet powerful mechanism for optimizations that can be made programatically, while relying on the standard AspectJ weaver.

5. Architecture and Implementation

In this section we describe the architecture of HotWave and explain the transformation steps performed upon runtime weaving.

Figure 6 depicts HotWave's architecture. The three main components have the following responsibilities.

1. *Agent:* Startup and initialization of HotWave.
2. *Aspect Manager:* Ensures that all affected classes are (re)woven with the most recent aspect provided by the user.
3. *Transformer:* Interaction with the standard AspectJ weaver and transformation of woven code to support inter-advice communication and weaving within the standard Java class library, and to ensure compatibility with code hotswapping constraints.

As described in Section 3, HotWave's programming model supports runtime weaving with an initially provided aspect, as well as (re)weaving while the application is already executing. Below we

(a) Common use of an around advice in AspectJ:

```
public aspect TimeAspect {
   pointcut allCalls() : call(* *.*(..)) && !within(TimeAspect);

   Object around() : allCalls() {
      long start = System.nanoTime();
      try {
         return proceed(); // proceed with the execution
      } finally {
         logExecTime(thisJoinPoint, System.nanoTime() - start);
      }
   }
   ...
}
```

(b) Equivalent aspect using inter-advice communication:

```
public aspect TimeAspect {
   pointcut allCalls() : call(* *.*(..)) && !within(TimeAspect);

   @InvocationLocal
   public static long start;

   before() : allCalls() { start = System.nanoTime(); }

   after() : allCalls() {
      logExecTime(thisJoinPoint, System.nanoTime() - start);
   }
   ...
}
```

Figure 4. Example of an `around` advice emulation

(a) Cflow example in AspectJ:

```
public aspect CflowAspect {
   before() : call(* *.*(..)) && cflow(call(* F.f())) &&
              !within(CflowAspect) {
      System.out.println(thisJoinPoint);
   }
}
```

(b) Efficient equivalent implementation with invocation-local variables:

```
public aspect CflowAspect {
   public static ThreadLocal<Integer> counter =
      new ThreadLocal<Integer>() {
         protected Integer initialValue() { return 0; }
   };

   @InvocationLocal
   public static int inCflow = 0;

   @InvocationLocal
   public static boolean isCached = false;

   before() : call(* F.f()) {
      counter.set(counter.get() + 1);
      System.out.println(thisJoinPoint);
   }

   after() : call(* F.f()) { counter.set(counter.get() - 1); }

   before() : call(* *.*(..)) && !call(* F.f()) &&
              !within(CflowAspect) {
      if (!isCached) {
         inCflow = counter.get(); // cache value
         isCached = true;
      }
      if (inCflow > 0) { System.out.println(thisJoinPoint); }
   }
}
```

Figure 5. Efficiently emulating a `cflow` pointcut

focus on the more complex case of (re)weaving in an already executing application.

Upon startup, HotWave's Agent is activated by the JVM, which invokes the Agent's `premain` method after JVM bootstrapping and before the application starts (① in Figure 6). The Agent initializes all HotWave components and registers the Transformer through the `java.lang.instrument` API ②. Subsequently, whenever a class is loaded or redefined, the Transformer receives a callback from the JVM.

When the user triggers (re)weaving with a new aspect ③, the Aspect Manager ensures (in a synchronized manner) that all affected classes are (re)woven, including previously loaded classes, classes that are being concurrently loaded respectively woven with a previous aspect, as well as classes that will be loaded later. To this end, the Aspect Manager informs the Transformer about the new aspect ④ and triggers the redefinition of previously loaded classes ⑤. To trigger (re)weaving, the new aspect may be provided either in compiled form or as AspectJ source code. In the latter case, the Aspect Manager compiles the aspect using the AspectJ compiler ⑥.

The Transformer receives `transform` requests both for classes to be redefined for hotswapping, as well as for newly loaded classes ⑦. These `transform` requests convey the original class bytes and the defining classloader. First, the Transformer weaves the original class bytes with the standard AspectJ weaver ⑧. Afterwards, the Transformer decides whether some of the following three bytecode transformations need to be applied to the woven code: (a) moving of static fields inserted by the AspectJ weaver into a separate class; (b) insertion of bypasses that allow threads to temporarily bypass woven code in a method; (c) advice inlining and local variable allocation for inter-advice communication.

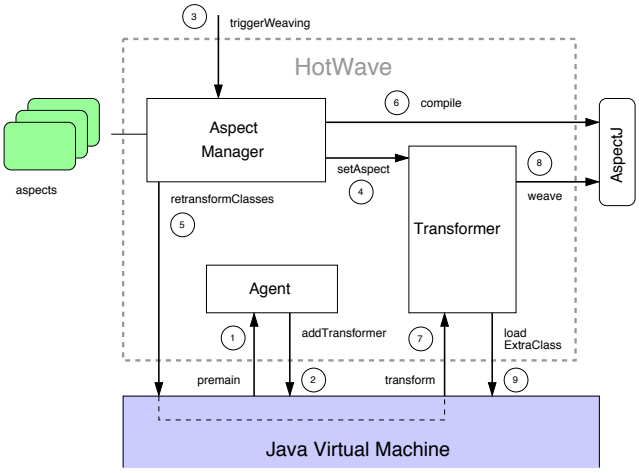

Figure 6. Architecture of HotWave

Transformation (a) is necessary to handle static fields introduced by the AspectJ weaver in support of join points.[6] As the insertion of static fields is not supported by Java's current hotswap-

[6] If an advice accesses a static or dynamic join point, for instance through `thisJoinPointStaticPart` respectively through `thisJoinPoint`, woven code passes the corresponding join point instance to the advice. Static join points are created in the woven static initializer and kept in static fields. Dynamic join points are allocated in woven code, but also refer to corresponding static join points.

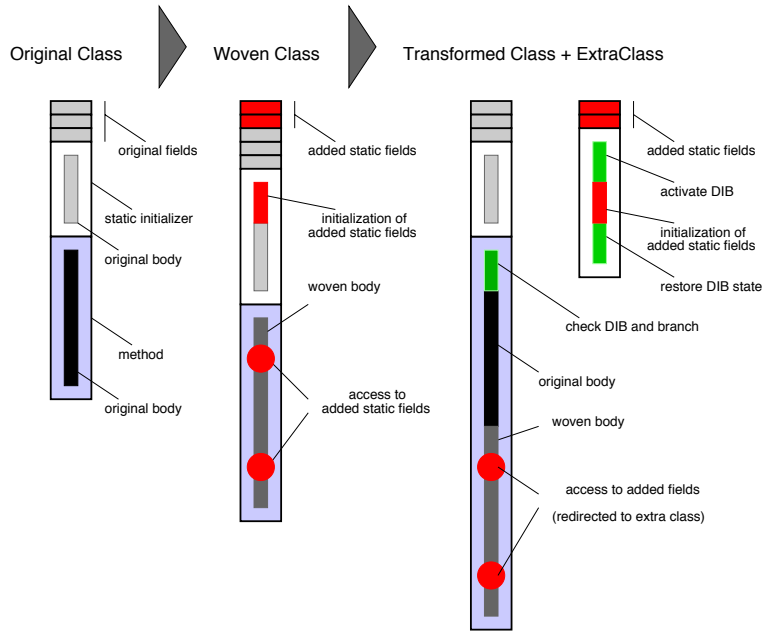

Figure 7. Code transformations: aspect weaving, moving inserted static fields into a new class, support for bypassing woven code

ping mechanism, the Transformer splits the woven class by moving the added static fields and the corresponding initialization code (within the woven static initializer) into a new class in the same package. If the moved static fields are private, their visibility is changed to package-visible. Access to the static fields in woven method bodies is redirected to the new class. Figure 7 illustrates this code transformation.

Since Java supports custom classloading, it is necessary to provide the introduced new class, called ExtraClass in Figures 6 and 7, to the defining classloader of the class under transformation (⑨ in Figure 6). Unfortunately, Java lacks a general mechanism for this purpose. While the `java.lang.instrument` API supports the addition of class archives in the search path of the bootstrap and system classloaders, there is no guarantee that a custom classloader follows the parent-delegation model suggested in the JDK, where a classloader first asks a parent classloader to find a class. For example, the dynamic module system OSGi [25] does not conform to the parent-delegation model. Consequently, the added class may not be found by a custom classloader, even though it could be located by the bootstrap or system classloader. To solve this problem, HotWave applies a patch to `java.lang.ClassLoader`, introducing a new method that forces the definition a given class. Since a well-behaved classloader must not define the same class multiple times (see [12], Section 12.2), it first checks whether a requested class has previously been defined. Hence, forcing a classloader to define an added class, normally ensures that the class will be found upon subsequent classloading requests. Our classloader patch has been successfully tested with various JDK versions and with many different applications, including OSGi-based systems, such as Eclipse. Note that in some JVMs, the bootstrap classloader is implemented in native code and represented by a null value. Hence, if the defining classloader is the bootstrap classloader (e.g., for classes in the `java.*` packages), the aforementioned patch may not be applicable. However, the mechanisms provided by the `java.lang.instrument` API suffice to make the bootstrap classloader find an added class.

Transformation (b) enables the bypassing of woven code. This feature is essential in woven classes within the Java class library, since methods in the Java class library are invoked during runtime weaving and may also be invoked in advices. In order to make runtime weaving transparent to the aspect and to avoid endless recursions, HotWave provides a mechanism that allows each thread to temporarily bypass woven code, reverting to the original code. This mechanism, called the Dynamic Inserted-Code Bypass (DIB), was first introduced in the FERRARI framework [7] and is also used in the MAJOR aspect weaver [34, 35]. Figure 7 illustrates the generated code in support of the DIB mechanism. Each thread has a thread-local flag indicating whether it shall bypass woven code. In a transformed method, first the state of that flag is checked to determine whether the original or the woven body shall be executed. Hence, the DIB mechanism relies on code duplication within method bodies. Furthermore, within the static initializer of an added class, the DIB mechanism is activated in order to ensure that the creation of static join point instances does not trigger any advice invocation.

Transformation (c) implements HotWave's inter-advice communication model. The advices accessing invocation-local variables are inlined and a local variable is allocated for each invocation-local variable used within a woven method body. Advice inlining is slightly complicated, because advice methods may take some arguments representing context information, such as static or dynamic join point instances. The Transformer's inlining algorithm determines the number and types of the advice arguments from the advice's method signature. The bytecodes that access the static fields corresponding to invocation-local variables are simply replaced with bytecodes for loading/storing from/in the local variables. Each invocation-local variable is initialized in the beginning of a transformed method with the value stored in the corresponding static field in the aspect, which is assigned only during execution of the aspect's static initializer. The Java memory model [12, 19] ensures that the value assigned by the static initializer is visible to all threads.

6. Case Study: Adaptive Profiling

Profiling is an important technique for performance analysis and for program understanding. There is a large body of related work on profiling [2, 11, 36], but prevailing approaches often depend on low-level, hard-coded instrumentations.

In *adaptive profiling*, the profiler code is adapted at runtime based on user choices, in order to restrict profiling to only part of an executing application or to enable respectively disable the collection of certain dynamic metrics. Adaptive profiling helps reduce profiling overhead, since only data is gathered that the user is currently interested in.

A good example of an adaptive profiler is JFluid [11], which has been integrated in the NetBeans Profiler [22]. JFluid measures execution time for selected methods and generates a Calling Context Tree (CCT) [2] to help analyze the contributions of direct and indirect callees to the execution time of the selected methods. When the user selects different methods for profiling at runtime, JFluid adapts the profiling code accordingly with the aid of code hotswapping. However, because JFluid relies on low-level bytecode instrumentation techniques, it cannot be easily extended; the set of dynamic metrics collected by JFluid is hard-coded.

In this section we present an `AdaptiveProfiler` aspect that leverages HotWave's distinguishing features, that is, runtime weaving for aspect adaptation, complete method coverage for gathering comprehensive profiling data, and inter-advice communication for efficiently passing state between advices. In contrast to prevailing adaptive profilers, the `AdaptiveProfiler` aspect is compact and can be easily extended, such as for the collection of additional dynamic metrics.

The `AdaptiveProfiler` aspect illustrated in Figure 8 creates a complete CCT covering all method execution in an application, including methods in the Java class library (for simplicity, we do not consider constructors in this example). It allows the user to select the profiling scope, that is, the set of methods for which dynamic metrics shall be collected. At any time, the profiling scope can be changed. In Figure 8, the meta-variable X represents the profiling scope. Typically, it is a logic formula using the AspectJ `within` and `withincode` constructs. For example, the following scope

```
X = within(java.util..*) ||
    ( within(java.lang.*) &&
      !withincode(* java.lang.Object.finalize()) )
```

selects the package `java.util`, including all subpackages, as well as the package `java.lang`, excluding the method `java.lang.Object.finalize()`.

In Figure 8, the type `AspectData` acts as shared data holder for all versions of the aspect that have been woven. It keeps the root of the CCT, which we assume to be a thread-safe datastructure shared between all threads, as well as the thread-local variable `currentNode` representing the current position in the CCT for each thread. Each node in the CCT is represented by an instance of type `CCTNode`, which stores the dynamic metrics collected for the corresponding calling context and offers a simple interface to update the profiling data in the form of methods `profile`M`(...)`, where M corresponds to a dynamic metric (`profileTime(long time)`, `profileAllocation(Object o)`, etc.). The method `profileCall(JoinPoint.StaticPart mid)` plays a special role; it returns the child CCT node representing a callee or creates such a node if it does not already exist. The argument `mid` (method identifier) is the static join point identifying the callee method.

The first three advices in Figure 8 deal with CCT creation without collecting dynamic metrics; these advices are woven into methods matching the pointcut `!metricScope`, where

```
// Shared data accessed by all woven versions of the aspect
public class AspectData {
   public static final CCTNode root = new CCTNode();

   public static final ThreadLocal<CCTNode> currentNode =
      new ThreadLocal<CCTNode>() {
         protected CCTNode initialValue() { return root; }
   };
}

public aspect AdaptiveProfiler {
   pointcut metricScope() : X; // X represents the profiling scope

   pointcut execs() : execution(* *.*(..)) &&
                      !within(AdaptiveProfiler);

   @InvocationLocal
   public static CCTNode caller, callee;

   @InvocationLocal
   public static long startTime;


   // Advices for CCT creation without collecting dynamic metrics:

   before() : execs() && !metricScope() {
      caller = AspectData.currentNode.get();
      callee = caller.profileCall(thisJoinPointStaticPart);
      AspectData.currentNode.set(callee);
   }

   after() returning() : execs() && !metricScope() {
      AspectData.currentNode.set(caller);
   }

   before() : handler(*)  && !metricScope()  {
      AspectData.currentNode.set(callee);
   }


   // Advices for CCT creation and measurement of elapsed wall clock time:

   before() : execs() && metricScope() {
      caller = AspectData.currentNode.get();
      callee = caller.profileCall(thisJoinPointStaticPart);
      AspectData.currentNode.set(callee);
      startTime = System.nanoTime();
   }

   after returning() : execs() && metricScope() {
      callee.profileTime(System.nanoTime() - startTime);
      AspectData.currentNode.set(caller);
   }

   after throwing() : execs() && metricScope() {
      callee.profileTime(System.nanoTime() - startTime);
   }

   before() : handler(*)  && metricScope()  {
      AspectData.currentNode.set(callee);
   }
   ...
}
```

Figure 8. Simplified adaptive profiling aspect that generates a CCT; elapsed wall clock time is measured only for methods in the given profiling scope X

`metricScope` is the profiling scope (i.e., the user selection of methods for which dynamic metrics shall be measured). The first advice is woven in method entries. It loads the caller's `CCTNode` instance from the thread-local variable, looks up the callee's `CCTNode` (`profileCall(...)`), and stores it into the thread-local variable. The fields `caller` and `callee` are declared as `@InvocationLocal`. That is, thanks to HotWave's inter-advice communication mechanism, both the caller node and the callee node can be efficiently accessed from local variables in other ad-

```
public aspect AllocProfiler {
   ... // same code as in AdaptiveProfiler

   pointcut allocScope() : Y; // Y represents the profiling scope

   pointcut allocs() : call(*.new(..)) && !within(AllocProfiler);

   after() returning(Object o) : allocs() && allocScope() {
      callee.profileAllocation(o);
   }
}
```

Figure 9. Extended aspect to collect also object allocation metrics in a given profiling scope Y

vices that are woven into the same method body. The second advice, woven before method return (`after() returning() : execution(* *.*(..))`), restores the caller's `CCTNode` into the thread-local variable; it only captures normal method completion. The third advice is woven in the begin of each exception handler. It restores the callee `CCTNode` in the thread-local variable, that is, the `CCTNode` representing the calling context where an exception is caught. This ensures that the thread-local variable is correctly updated also in the case of abnormal method completion.[7]

The following four advices deal with CCT creation and dynamic metrics collection in the profiling scope. In the presented sample code, we focus only on elapsed wall clock time. The advices are very similar to the previously discussed advices, since both groups of advices have to maintain the CCT. In addition, the advices for the `metricScope` obtain the current time upon method entry and upon normal and abnormal method completion. The time read upon method entry is preserved in an invocation-local variable, allowing to compute the elapsed time upon method completion.

If the user changes the profiling scope, the aspect is updated (the meta-variable X is changed accordingly and the aspect name is updated), recompiled, and rewoven, resulting in hotswapping of some previously loaded classes. Usually, it is not necessary to reweave and hotswap all previously loaded classes. If the previous profiling scope described a set of methods X_{old} and the new profiling scope specifies a set of methods X_{new}, then only the classes that have a method in the symmetric difference $X_{old} \Delta X_{new}$ need to be rewoven and hotswapped. The remaining classes can continue using the previous aspect version. HotWave helps avoid unnecessary reweaving, because it allows to restrict the set of (previously loaded) classes that are rewoven with a new aspect and hotswapped. Newly loaded classes are always woven with the new aspect at load-time.

In order to highlight the flexibility of our AOP-based approach to tool development, Figure 9 illustrates an extension of the `AdaptiveProfiler` aspect, called `AllocProfiler`, in order to collect also object allocation metrics in a given profiling scope Y (which may be different from the profiling scope X). Thanks to HotWave, this profiler extension can be deployed at runtime, while an application is being profiled. Note that all versions of the `AdaptiveProfiler` and `AllocProfiler` aspects use the same shared data model, provided by the type `AspectData`. This is essential, as multiple aspect versions may be in use at the same time.

7. Conclusion

Dynamic AOP, which enables aspect adaptation at runtime, is an important feature to build advanced aspect-based software engineering tools, such as adaptive profilers. For such tools, it is also important that aspects can we woven into all classes, including the standard Java class library, in order to gather dynamic metrics that represent overall program execution. Unfortunately, prevailing approaches to dynamic AOP in Java often restrict the aspect language, rely on native code, or exclude certain system classes from aspect weaving.

In this paper, we introduced HotWave, a new aspect weaver that reconciles dynamic AOP, compatibility with AspectJ and with standard JVMs, and comprehensive aspect weaving into all classes. Furthermore, HotWave offers a unique mechanism, inter-advice communiation, for efficiently passing data between advices using local variables. As case study, we have shown that a sophisticated adaptive profiler can be concisely specified as an aspect for HotWave.

[7] An alternative approach would be to directly capture abnormal method completion with an `after() throwing()` pointcut. However, the AspectJ weaver would then insert an exception handler spanning the woven method, which may cause higher overhead in some JVMs.

Acknowledgment

The work presented in this paper has been supported by the Swiss National Science Foundation.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, B. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, N. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] G. Ammons, T. Ball, and J. R. Larus. 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*, pages 85–96. ACM Press, 1997.

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[4] L. D. Benavides, R. Douence, and M. Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 183–202, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[5] A. Bergel. FacetS: First class entities for an open dynamic AOP language. In *Proceedings of the Open and Dynamic Aspect Languages Workshop*, Mar. 2006.

[6] W. Binder and J. Hulaas. Flexible and efficient measurement of dynamic bytecode metrics. In *Fifth International Conference on Generative Programming and Component Engineering (GPCE-2006)*, pages 171–180, Portland, Oregon, USA, Oct. 2006. ACM.

[7] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.

[8] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009. `http://dx.doi.org/10.1002/spe.890`.

[9] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD*, pages 83–92, 2004.

[10] S. Chiba, Y. Sato, and M. Tatsubori. Using Hotswap for Implementing Dynamic AOP Systems. In *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[11] 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, pages 139–150. ACM Press, 2004.

[12] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

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

[14] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[15] JBoss. Javassist Project. Web pages at `http://jboss.com/javassist/`.

[16] JBoss. Open source middleware software. Web pages at `http://labs.jboss.com/jbossaop/`.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. 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*, pages 327–353, 2001.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[19] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[20] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction, volume 2622 of Springer Lecture Notes in Computer Science*, pages 46–60. Springer, 2003.

[21] P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 151–160, Savannah, GA, USA, 2009. ACM.

[22] NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`.

[23] A. Nicoara and G. Alonso. Making applications persistent at runtime. *International Conference on Data Engineering*, 15:1368–1372, 2007.

[24] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, New York, NY, USA, 2008. ACM.

[25] OSGi Alliance. OSGi Alliance Specifications. Web pages at `http://www.osgi.org/Specifications/`.

[26] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: An Aspect-based Distributed Dynamic Framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[27] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[28] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[29] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[30] Sun Microsystems, Inc. Java Platform Debugger Architecture (JDPA). Web pages at `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[31] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`.

[32] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive Programming in JAsCo. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM.

[33] A. Vasseur. Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz address it? In *Dynamic Aspects Workshop (DAW04)*, Lancaster, England, Mar. 2004.

[34] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, New York, NY, USA, Sept. 2008. ACM.

[35] A. Villazón, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*, pages 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.

[36] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.