Aspect Weaving in Standard Java Class Libraries

Alex Villazón
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
alex.villazon@lu.unisi.ch

Walter Binder
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
walter.binder@unisi.ch

Philippe Moret
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
philippe.moret@lu.unisi.ch

ABSTRACT

Aspect-oriented programming (AOP) has been successfully applied to application code thanks to techniques such as Java bytecode instrumentation. Unfortunately, with current technology, such as AspectJ, aspects cannot be woven into standard Java class libraries. This restriction is particularly unfortunate for aspects that would benefit from a complete bytecode coverage, such as profiling or debugging aspects. In this paper we present an adaptation of the popular AspectJ weaver that is able to weave aspects also into standard Java class libraries. We evaluate our approach with existing profiling aspects, which now cover all bytecode executing in the virtual machine. In addition, we present a new aspect for memory leak detection that also benefits from our approach.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.3.3 [**Language Constructs and Features**]: Frameworks

General Terms

Algorithms, Languages, Measurement

Keywords

Aspect-oriented programming, aspect weaving, bytecode instrumentation, profiling, memory leak detection

1. INTRODUCTION

Aspect-oriented programming (AOP) [18] is a powerful approach enabling a clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, monitoring and logging, and debugging support. AspectJ [17] provides AOP capabilities for Java, allowing new functionality to be systematically added to existing programs.

PPPJ 2008, September 9–11, 2008, Modena, Italy.

Java bytecode instrumentation is a widely used technique for profiling and, more generally, for any transformation in support of aspect weaving [26]. Based on bytecode instrumentation, aspect weaving tools, such as AspectJ [17] or *abc* [2], allow the insertion of code at well defined points in Java programs without resorting to source code manipulation.

Unfortunately, current aspect weavers do not support aspect weaving into classes of the JDK which strongly restricts the applicability of AOP. For example, building profilers with aspects is a promising approach [22], but the impossibility of weaving aspects into JDK classes causes incomplete profiles, limiting the practical value of this approach.

In general, instrumentation of the JDK is difficult and may crash the JVM, because of its sensitivity to modifications, notably during JVM bootstrapping. While aspect support can be integrated at the JVM level [23], such an approach is restrictive because it prevents the reuse of existing AOP tools. In addition, using modified JVMs and native code makes it difficult to leverage standard, state-of-the-art JVM and AOP technologies.

In prior work we introduced FERRARI [5] (*Framework for Exhaustive Rewriting and Reification with Advanced Runtime Instrumentation*), a generic bytecode instrumentation framework supporting the instrumentation of the whole JDK including all core classes, as well as dynamically loaded classes. FERRARI is neither an application-specific framework nor a low-level bytecode instrumentation toolkit. Instead, it generates the necessary program logic to enable general-purpose instrumentation of JDK and application classes, while giving advanced support for bootstrapping with an instrumented JDK. FERRARI provides a flexible interface enabling different user-defined instrumentations (UDIs) written in pure Java to control the instrumentation process.

In this paper, we present an *AspectJ UDI* which adapts existing and unmodified aspect weaving tools so as to weave aspects also into JDK classes. We show the soundness and potential of our approach with a sample aspect for memory leak detection. We describe the implementation of the AspectJ UDI and discuss its current limitations. In addition, we evaluate and validate our approach by weaving existing profiling aspects into the JDK.

As contributions, in this paper we highlight the benefits of comprehensive aspect weaving, covering also the standard Java class library. Moreover, we present an aspect for memory leak detection that benefits from JDK weaving. Additionally, we show that our approach is compatible with

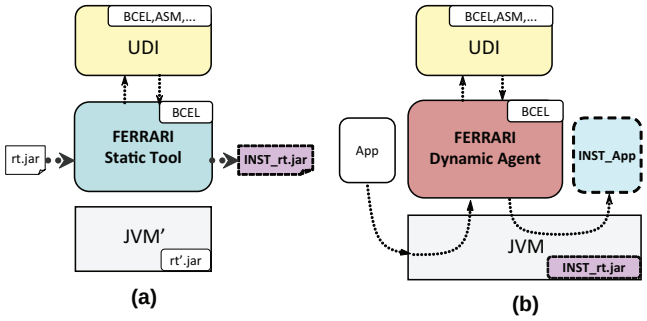

Figure 1: (a) Static and (b) dynamic instrumentation with FERRARI.

existing AspectJ tools and enables existing aspects to be applied to the JDK. The latter part of our evaluation is based on a set of profiling aspects published by others [22].

This paper is structured as follows. Section 2 summarizes the features provided by FERRARI, our generic instrumentation framework. Section 3 discusses the implementation of the AspectJ UDI and its current limitations. Section 4 shows an aspect for memory leak detection. Section 5 presents our evaluation results. Section 6 addresses related work. Finally, Section 7 concludes this paper.

2. THE FERRARI FRAMEWORK

FERRARI [5] consists of a static instrumentation tool and a runtime instrumentation agent. These tools are controlled by a UDI. 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-intrumented class bytes. FERRARI's general purpose API [5] allows the seamless integration of existing bytecode transformation tools through UDIs.

FERRARI provides a tool to statically instrument all JDK classes according to a given UDI. Figure 1(a) shows the static instrumentation of JDK classes (`rt.jar`), resulting in an instrumented JDK `INST_rt.jar` that is used by the JVM executing the application under dynamic instrumentation (see Figure 1(b)).

Application classes are dynamically instrumented by FERRARI's agent in collaboration with the UDI. The agent is based on the `java.lang.instrument` package introduced in JDK 1.5, ensuring portability. Figure 1(b) shows how application classes (`App`) are instrumented dynamically so that their instrumented versions (`INST_App`) are those actually linked by the JVM.

The current implementation of FERRARI uses Apache's bytecode engineering library BCEL [8]. UDIs are nevertheless free to use any bytecode engineering library, such as BCEL, ASM [21], Javassist [7], Soot [28], etc.

FERRARI offers generic mechanisms to ensure complete instrumentation 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.

```
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 2: Dynamic Inserted-code Bypass (DIB).

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

Figure 3: Activating the DIB for the current thread.

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 the instrumented code during the boostrapping of the JVM. For details, see [5].

Concerning issue (2), FERRARI introduces a thread-local flag, the *Dynamic Inserted-code Bypass* (DIB), which allows per-thread bypassing of the 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 2).

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 [5]. This support is only available for the static instrumentation of the JDK.

In general, methods invoked by UDI-inserted code must not themselves execute any UDI-inserted code, which otherwise could result in infinite recursions. Notably, 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` can be used as shown in Figure 3, which allows the calling thread to temporarily bypass intrumented-code at runtime. This issue is particularly relevant for our AspectJ UDI and will be explained in more detail later.

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.

FERRARI's instrumentation agent, the UDI, UDI-runtime-classes, and bytecode engineering library classes are excluded from normal instrumentation. All other classes are instrumented either statically or dynamically at load-time.

3. THE ASPECTJ UDI

In this section we discuss how we adapted existing AspectJ tools as a FERRARI UDI. The new AspectJ UDI allows aspect developers to weave aspects into applications and also into JDK classes. No modifications of AspectJ tools were necessary. To our knowledge, this is the first successful attempt of applying aspects to the standard Java class library in a portable way. This enables the creation of new tools for crosscutting concerns and the application of existing aspects to JDK classes. In the following, we concentrate on the integration of AspectJ with FERRARI and discuss the limitations of weaving aspects into the JDK.

3.1 Aspect Weaving

Before describing the AspectJ UDI implementation, we introduce the most important concepts and describe different transformations made during the weaving process and their relation to FERRARI features. In this section we refer to the bytecode-level instrumentation made by the AspectJ weaver [15].

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

Aspects are compiled into standard Java classes. Aspect classes are used during the weaving process and also at runtime; they hold the weaving rules (i.e., pointcuts), and they implement the actions associated with them (i.e., advices). AspectJ supports three kinds of advices: `before`, `after`, and `around`, which are executed prior, following, or surrounding a join point's execution. In the aspect class, advices are compiled into methods. In the woven class, the weaver inserts code to invoke these advice methods. Aspect classes are used by the AspectJ UDI to weave application and JDK classes. Moreover, the advice methods are invoked by instrumented code, i.e., aspect classes are UDI-runtime-classes.

The execution of an advice implies the creation of an aspect instance. By default, aspect instances are singletons and are accessed by the woven classes through the `aspectOf()` static method that is generated by the weaver in the aspect class. Because the invocations to `aspectOf()` and advice methods are made from instrumented code, the UDI must ensure that these methods do not themselves execute instrumented code in order to prevent infinite recursions. An automated tool applies the code pattern shown in Figure 3 to every constructor, static initializer, advice method, and generated method in the aspect class, in order to temporarily bypass the execution of instrumented code.

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. All method signatures in the woven class remain unchanged which simplifies the integration of AspectJ with FERRARI.

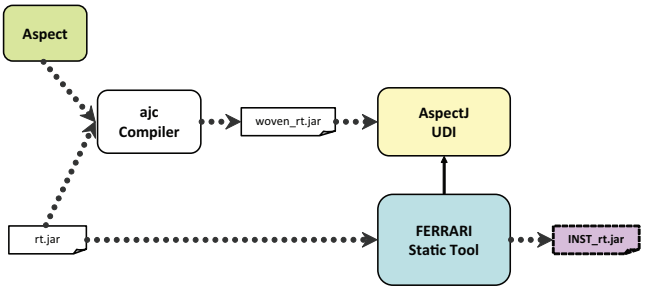

Figure 4: Static JDK instrumentation with the AspectJ UDI: First, `ajc` weaves the aspect into the JDK (`woven_rt.jar`). Afterwards, FERRARI inserts the bypasses in the woven JDK (`INST_rt.jar`).

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), 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 AspectJ UDI uses this feature for JDK weaving.

Note that AspectJ also supports non-singleton aspect instances through special `per*` clauses (e.g., for per-object or per-control flow aspect associations). Unfortunately, this mechanism modifies the class hierarchy of the woven classes (to implement a compiler generated interface). Changing the hierarchy of JDK classes may break the bootstrapping and therefore our current AspectJ UDI does not support constructs that modify the class hierarchy for JDK weaving. Other transformations (e.g., method body modifications, insertion of advice invocation, additional static fields, access to reflective information in advices) are supported for JDK weaving.

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. Our AspectJ UDI currently does not support static-crosscutting for JDK weaving, though we are working on relaxing this restriction.

3.2 Adapting AspectJ Tools

The current AspectJ implementation 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 for bytecode weaving. AspectJ supports two different approaches for load-time weaving: the

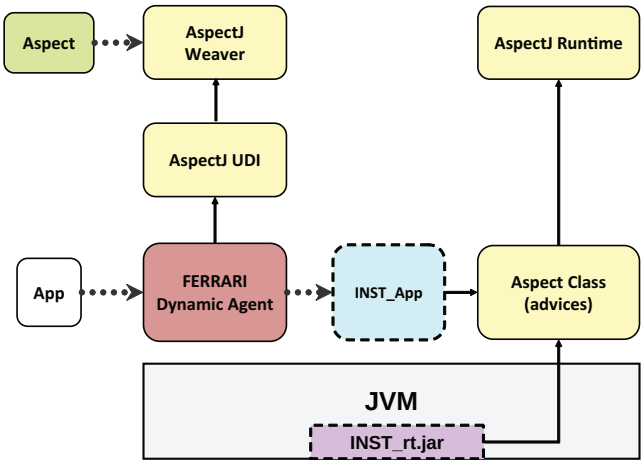

Figure 5: Dynamic instrumentation with the AspectJ UDI.

first one uses a customized classloader and the second one is based on the `java.lang.instrument` API.

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. Unfortunately, the weaving adaptor does not support weaving within the JDK and therefore cannot be used by FERRARI's static instrumentation tool. There is no such restriction on the `ajc` compiler and weaver. Therefore, for the JDK, we use a two-phase instrumentation: first we use the `ajc` bytecode weaver to weave the aspect into the JDK (see Figure 4 on the left side), and in the second phase, the AspectJ UDI uses the woven JDK such that FERRARI can insert the bypasses (see Figure 4 on the right side).

During the second phase, since no method signatures are modified by the weaver, the AspectJ UDI performs a rather simple sequence: For each class to instrument, it reads the already woven version (rather than 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 AspectJ UDI moves added static fields and the corresponding parts of the static initializers into extra classes.[1] This information is returned to FERRARI. The resulting `INST_rt.jar` in Figure 4 has the woven classes with the bypasses together with the extra classes.

For dynamic instrumentation, the AspectJ UDI uses the AspectJ weaver through the `WeavingAdaptor`. FERRARI's runtime instrumentation agent plays a similar role as AspectJ's load-time mechanisms. Once a class is woven, the AspectJ UDI determines which methods were modified by the weaver, and tells FERRARI to generate bypass code to enable callbacks from advices into application code.

[1]Thanks to the JVM's lazy class initialization, 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.

Figure 5 illustrates the execution of a dynamically woven application running on top of a woven JDK. In this example we assume that the same aspect was woven in the application and in the JDK (which is not necessarily always the case, as different aspects can be woven into the JDK and the application). Both the woven JDK and application code (`INST_rt.jar` and `INST_App`) invoke the advices on the aspect class, i.e., the UDI-runtime-class, which uses AspectJ's runtime system for full dynamic aspect support. Thus, FERRARI and the AspectJ UDI provide advanced support for JDK weaving which is complementary to existing AspectJ tools.

4. WEAVING ASPECTS INTO THE JDK

In this section, we describe a sample aspect for memory leak detection enabling simple leak analysis without resorting to low-level profiling tools. We give a detailed description of the aspect and an example of its use.

4.1 Aspect for Memory Leak Detection

Despite of automated memory management, Java programs can still suffer from memory leaks resulting from unintentional object retention. Detecting memory leaks can be difficult; typically, developers have to use profilers for digging information about loitering objects on the heap. Such profilers use a native profiling API of the underlying JVM, such as the JVMPI [24] or the more recent JVMTI [25], or rely on a modified JVM [16, 9].

Some attempts have been made to build memory leak detection tools based on aspects [19]. In such an approach, the application code is woven with an aspect capturing constructor calls, field assignments to track references between objects, and method invocations passing objects across package boundaries. Unfortunately, such an approach fails to capture potential memory leaks if object allocation is implicit (no `new` instruction) in the application code as is the case in the code excerpt[2] shown in Figure 6. In this example, the method `slowlyLeakingVector()` simply adds and removes elements from a vector. However, the remove operation leaves objects in the vector which cannot be garbage collected, representing a potential memory leak (the allocation is actually performed by the `java.lang.Integer` class and therefore cannot be intercepted at the application level). By enabling aspects to be woven into the Java class library, our approach solves this limitation allowing to identify such potential memory leaks.

In the following we present an aspect for memory leak detection, `LeakDetectorAspect`, and show how the memory leak in Figure 6 is found.

4.2 The LeakDetectorAspect Implementation

The main idea of the `LeakDetectorAspect` is to capture all object allocation sites and to keep a weak reference to every allocated object together with additional information about the allocation context. When the garbage collector reclaims unused objects, the weak references are removed.

[2]This method is part of the sample code used in `http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/` to show how to detect memory leaks with IBM Rational Application Developer, which uses the LeakBot [20] technology based on the JVMPI.

```
static Vector myVector = new Vector();
...

public void slowlyLeakingVector(int iter, int count) {
    for (int i=0; i<iter; i++) {
         for (int n=0; n<count; n++)
              myVector.add(Integer.toString(n+i));
         for (int n=count-1; n>0; n--) {
              // Oops, it should be n>=0
              myVector.removeElementAt(n);
         }
    }
}
```

Figure 6: Example of a memory leak in Java.

The remaining referenced objects correspond to potential memory leaks.

Figure 7 shows a simplified version of the `LeakDetectorAspect`. The `allAllocs` pointcut (lines 2–3) specifies that every object allocation join point (i.e., calls to `new`) must be captured.[3] The `!within()` pointcut designator prevents the aspect itself to be woven to avoid infinite recursions.[4]

The advice associated with the `allAllocs` pointcut is shown in lines 11–19. We use the `after returning` construct to get the reference to the newly created object, which is passed as the `newObject` argument to the advice. We create a weak reference to `newObject` with additional information about the allocation context. To this end, we use the `java.lang.ref` API which provides three reference types (`SoftReference`, `WeakReference`, and `PhantomReference`). These reference types do not prevent the referenced object from being garbage collected. We have chosen the `PhantomReference` type, because in contrast to the others, the referenced objects reclaimed by the garbage collector are "cleared" (i.e., set to `null`) only after finalisation. This ensures that our aspect does not miss any potential memory leaks.

The `MLRef` class extends `PhantomReference` with two important informations to locate the source of memory leaks (see lines 32–41). The first one is provided by the reflective AspectJ API through the `JoinPoint.StaticPart`, which keeps information about the join point, such as the location of the join point in the source. The second one allows us to keep more complete information about the calling context to be able to locate the actual source of the memory leak. To this end, we store a `Throwable` instance with the full stack trace.[5]

The advice code in lines 14–17 shows how the weak reference to `newObject` is created by using the special predefined variable `thisJoinPointStaticPart`, and the filled stack trace with the new `Throwable` object. The `ReferenceQueue` instance passed as last argument is used by the garbage collector to place weak references that are cleared. We store all non-cleared weak references in the `refs` set. Since the advice can be invoked concurrently by several threads, `refs` is `synchronized`.

As discussed in Section 3, the invocation of the advice happens within instrumented code and the AspectJ UDI ensures that no invocation to instrumented code happens within any advice method body. In lines 12, 13, and 18 we show the generated code pattern to avoid infinite recursion. The same code pattern can be seen in the aspect constructor (lines 22, 23, and 29), and is also added to the compiler-generated methods, such as the `aspectOf()` static method and the static initializer (not shown).

Two threads are used by the aspect. (1) The `CleanupThread` (lines 71–80) constantly removes all cleared weak references from the `refs` set. This thread is started as a daemon thread in the aspect constructor (lines 27–28). (2) The `ShutdownThread` is responsible for the final cleanup and the actual memory leak analysis. We use a JVM *shutdown hook* to trigger the execution of this thread after all non-daemon threads have terminated. Firstly, the garbage collection is forced until the free available memory remains stable, and, in order to avoid an infinite loop in case a daemon thread continues allocating memory, we enforce the upper limit `MAXGC` on the number of iterations (lines 45–56). Secondly, we poll the `refQueue` and remove the cleared weak references from the `refs` set (lines 58–62). The final loop (lines 64–67) performs the actual leak analysis. Note that the methods in these thread classes do not need to activate the DIB. As the threads are created while the DIB is active, they "inherit" the activated DIB state as explained in Section 2.

4.3 Finding Memory Leaks

Lets come back to the example of Figure 6. Firstly, we wove the `LeakDetectorAspect` only into application code using the standard AspectJ load-time weaver. We invoked the method `slowlyLeakingVector(int iter, int count)` with arguments `iter=1000` and `count=10`, which should normally result in 10000 allocated objects and 1000 non-reclaimed objects. The aspect however reported only 1 object that was not reclaimed (which corresponds to the `Vector` instance of Figure 6 allocated in the static initializer), out of a total of 12 allocated objects. Of course, this result is incomplete because there is no allocation bytecode in the method.

Secondly, we wove the aspect into both the JDK and into application classes using FERRARI and the AspectJ UDI. Now the aspect reported the correct information of non-reclaimed objects. Figure 8 shows part of the analysis result. The aspect reported a total of 20157 allocated objects and 2042 non-reclaimed objects. We observe that as expected, there are 1000 occurrences of non-reclaimed `String` objects. In addition, 1000 non-reclaimed character arrays (`char[](int)`) were reported as created by `toString()`; i.e., for each entry in the vector, two objects are allocated.

In addition, we have precise information about the source of the potential memory leaks. By analyzing the stack traces in Figure 8, we can see that the problem stems from the `slowlyLeakingVector()` method in `LeakSample.java` at `line 13`. The aspect also reports 42 additional non-reclaimed objects, including objects created by the static

[3]Array allocations can be captured by using the `joinpoints:arrayconstructor` extension of the AspectJ weaver. We always enable this extension.

[4]FERRARI's DIB mechanism is complementary to AspectJ's `within` compile-time instrumentation bypass; the DIB enables selective, dynamic, per-thread bypassing of UDI-inserted code.

[5]Note that this mechanism is rather expensive compared to other approaches, such as using instrumentation for calling context reification [4] and passing the context as additional argument rather than filling the stack trace for every allocated object. However, here we want to show that it is possible to build simple tools using standard mechanisms.

```
1  public final aspect LeakDetectorAspect {
2      pointcut allAllocs() : call(*.new(..))
3                          && !within(LeakDetectorAspect);
4
5      static private final Set<MLRef> refs =
6             Collections.synchronizedSet(new HashSet<MLRef>());
7      static private final ReferenceQueue<Object> refQueue =
8             new ReferenceQueue<Object>();
9      static private final int MAXGC = ...
10
11     after() returning(Object newObject) : allAllocs() {
12         boolean oldValue = DIB.getFlagAndSetTrue(); // generated
13         try {                                       // generated
14             MLRef objref = new MLRef(thisJoinPointStaticPart,
15                            new Throwable().fillInStackTrace(),
16                            newObject, refQueue);
17             refs.add(objref);
18         } finally { DIB.setFlag(oldValue); }        // generated
19     }
20
21     public LeakDetectorAspect() {
22         boolean oldValue = DIB.getFlagAndSetTrue(); // generated
23         try {                                       // generated
24             ShutdownThread shutdown = new ShutdownThread();
25             Runtime.getRuntime().addShutdownHook(shutdown);
26
27             CleanupThread cleanup = new CleanupThread();
28             cleanup.setDaemon(true);  cleanup.start();
29        } finally { DIB.setFlag(oldValue); }         // generated
30     }
31
32     private static final class MLRef
33                             extends PhantomReference<Object> {
34         final JoinPoint.StaticPart sjp;
35         final Throwable thrw;
36
37         MLRef(JoinPoint.StaticPart s, Throwable t,
38               Object o, ReferenceQueue<Object> q) {
39             super(o,q);  sjp = s;   thrw = t;
40        }
41     }
42
43     private static final class ShutdownThread extends Thread {
44         public void run() {
45             Runtime rt = Runtime.getRuntime();
46             long mem, oldmem;
47             int countgc = 0;
48             mem = rt.freeMemory();
49
50             do {
51                 oldmem = mem;
52                 rt.gc();
53                 System.runFinalization();
54                 countgc++;
55                 mem = rt.freeMemory();
56             } while(mem!=oldmem && countgc < MAXGC);
57
58             MLRef objref = (MLRef)refQueue.poll();
59             while(objref != null) {
60                 refs.remove(objref);
61                 objref = (MLRef) refQueue.poll();
62             }
63
64             synchronized(refs) {
65                 for(MLRef oref : refs)
66                     doLeakAnalysis(oref); // not shown here
67             }
68         }
69     }
70
71     private static final class CleanupThread extends Thread {
72         public void run() {
73             while(true) {
74                 try {
75                     MLRef objref = (MLRef)refQueue.remove();
76                     refs.remove(objref);
77                 } catch(InterruptedException e) { }
78             }
79         }
80     }
81 }
```

Figure 7: Simplified `LeakDetectorAspect`.

initializer of the application class (e.g., the `Vector` instance itself and its internals) and other objects internally created by the JVM (e.g., created by `java.lang.Thread.<init>`, `java.lang.Shutdown.shutdown()`, etc.).

The `LeakDetectorAspect` is only a straw man to illustrate the benefits of aspect weaving in the JDK. In fact, memory leaks due to objects that are referenced only by non-daemon threads (and not by any static fields) will not be reported, because they can be reclaimed before the leak analysis. However, the aspect can be easily changed to perform the leak analysis at any moment during program execution.

5. EVALUATION

In this section we present two different evaluations. Firstly, we apply the `LeakDetectorAspect` to standard benchmarks and compare the number of allocated and non-reclaimed objects, with and without woven JDK. Secondly, we evaluate the execution time overhead using existing profiling aspects and measure the code bloat due to aspect weaving and FERRARI's code duplication for the bypasses.

5.1 Settings

For our evaluation, we use two benchmark suites, SPEC JVM98[6] (problem size 100) and DaCapo[7] (version 'dacapo-2006-10-MR2'; default workload size). JVM98 consists of 7 benchmarks, whereas DaCapo, which is intended to provide more realistic workloads than JVM98 [6], has 11 benchmarks.

Our test platform is a Linux Fedora Core 2 computer (Intel Pentium 4, 2,66 GHz, 1024 MB RAM). The metric used for DaCapo and JVM98 is the execution time in seconds. We present measurements made with the Sun JDK 1.7.0-ea-b25 HotSpot Client VM.

5.2 Memory Leak Aspect Evaluation

We applied the `LeakDetectorAspect` to the JVM98 benchmark suite, using both the standard AspectJ weaver, as well as FERRARI with the AspectJ UDI. We extended the aspect to report "application only" allocations, by filtering out those allocations that did not include any stack frame of an application method. For example, in Figure 8 the traces including `LeakSample.main` and `LeakSample.<clinit>` are evaluated, whereas other traces that have no methods from the application class are filtered out. Here we do not perform any memory leak analysis of JVM98; a memory leak analysis of JVM98 and DaCapo can be found in [16].

Table 1 reports the number of object allocations ('Alloc.') and non-garbage collected objects ('N-GC') when running the application with the standard AspectJ weaver (setting 'AspectJ weaver'; unmodified JDK) respectively with FERRARI and the AspectJ UDI (setting 'FERRARI + AJ UDI'; woven JDK).

We calculated the percentage of the values reported in the setting 'AspectJ weaver' with respect to those reported in the setting 'FERRARI + AJ UDI'. This gives us a measure of the actual "coverage" of the aspect without weaving the JDK ('Cov.' in Table 1).

We observe that the aspect is able to cover a high percentage of allocated objects for 'jess' and 'mtrt' (99.60%

[6] `http://www.spec.org/osg/jvm98/`
[7] `http://www.dacapo-bench.org/`

```
Total number of allocated objects:          20157
Total number of non-reclaimed objects:       2042

Details of non-reclaimed objects (potential memory leaks):

Number of occurrences: 1000
Location and stack trace: constructor-call  java.lang.String(int, int, char[]) Integer.java:308
=> LeakSample.main(LeakSample.java:51)
  => LeakSample.slowlyLeakingVector(LeakSample.java:13)
    => java.lang.Integer.toString(Integer.java)

Number of occurrences: 1000
Location and stack trace: constructor-call  char[](int) Integer.java:306
=> LeakSample.main(LeakSample.java:51)
  => LeakSample.slowlyLeakingVector(LeakSample.java:13)
    => java.lang.Integer.toString(Integer.java)

Number of occurrences: 1
Location and stack trace: constructor-call  java.util.Vector() LeakSample.java:7
=> LeakSample.<clinit>(LeakSample.java: 7)

Number of occurrences: 3
Location and stack trace: constructor-call  char[](int) String.java:2726
=> java.lang.Thread.<init>(Thread.java)
  => java.lang.Thread.init(Thread.java)
    => java.lang.String.toCharArray(String.java)

Number of occurrences: 2
Location and stack trace: constructor-call  java.lang.Object() Thread.java:225
=> java.lang.Thread.<init>(Thread.java)

...
```

Figure 8: Part of the leak analysis reported by the `LeakDetectorAspect`.

and 97.28%). However, the coverage for non-garbage collected objects is rather low for all benchmarks. This results in incomplete information for memory leak analysis. We also observe that the 'db' benchmark has the lowest allocation coverage of only 4.84%, which can be explained by the fact that most object allocations happen in JDK classes [14] and therefore cannot be intercepted without a woven JDK.

In [6] the authors use a modified version of the Jikes RVM [1] in order to report static metrics, dynamic metrics, and performance results for the DaCapo and JVM98 benchmarks. Amongst others, they report the number of allocated and live objects. Even though we cannot make a strict comparison with our results, because of the use of different JDKs and JVMs, we obtained similar values for 'jess', 'db', 'javac', and 'mtrt' with a relative difference (by taking our measurements as reference) between 0.7% and 0.9%. For 'compress', 'mpegaudio', and 'jack', the relative difference is between 49.8% and 57.9%. In our approach, these results were obtained without resorting to any JVM modification.

5.3 Profiling with the AspectJ UDI

To show the applicability to existing aspects, we evaluated our AspectJ UDI using DJProf[8], a set of aspects for profiling heap usage, average object lifetime, average object "waste" time (object lifetime minus its useful time), time spent in each method invocation, and the number of method calls. In the following we make reference to these aspects as `heap`, `lifetime`, `waste`, `cpu`, and `call-count`. As the authors of DJProf witnessed [22], the major limitation of DJProf was the inability to weave the profiling aspects into the JDK, resulting in incomplete profiles. FERRARI and the AspectJ UDI solve exactly that problem.

[8] `http://www.mcs.vuw.ac.nz/~djp/djprof/`

	AspectJ weaver				FERRARI + AJ UDI	
	Alloc.	**Cov.**	**N-GC**	**Cov.**	**Alloc.**	**N-GC**
JVM98						
compress	836	16.07 %	34	2.52 %	5,201	1,348
jess	7,902,648	99.60 %	270	8.57 %	7,934,455	3,151
db	155,296	4.84 %	9	0.70 %	3,210,242	1,284
javac	3,740,232	62.78 %	5,084	33.51 %	5,957,607	15,173
mpegaudio	1,386	19.29 %	1,290	48.48 %	7,186	2,661
mtrt	6,457,760	97.28 %	13	0.87 %	6,638,029	1,486
jack	1,435,873	22.91 %	5,729	37.85 %	6,268,748	15,138

Table 1: Comparison of the `LeakDetectorAspect` using the AspectJ weaver respectively FERRARI and the AspectJ UDI.

In the DJProf aspects we corrected some race conditions (e.g., by using atomic integers for counters that are shared between threads), but we did not change the structure of the aspects, since our goal is to show that our framework works well with existing aspects.

Although we successfully applied the `waste` aspect to all benchmarks, we excluded `waste` from our evaluation, because of its excessive overhead [22]. Alternative and more efficient implementations of this aspect are available in DJProf, but unfortunately they modify the class hierarchy, preventing JDK weaving.

The `lifetime` and `cpu` profiling aspects use a configurable sampling mechanism to reduce overhead. In our configuration, only every 100th object is profiled with the `lifetime` aspect, and the sampling interval is set to 100ms for the `cpu` aspect.

Table 2 shows our measurements and the overhead for each profiling aspect. Each measurement corresponds to the median of 15 runs within the same JVM process. In the

	Orig.	'heap' aspect				'lifetime' aspect				'cpu' aspect				'call-count' aspect			
		AspectJ weaver		FERRARI + AJ UDI		AspectJ weaver		FERRARI + AJ UDI		AspectJ weaver		FERRARI + AJ UDI		AspectJ weaver		FERRARI + AJ UDI	
DaCapo	[s]	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**
antlr	3.31	5.05	**1.53**	7.45	**2.25**	4.42	**1.34**	5.10	**1.54**	6.77	**2.05**	19.60	**5.92**	13.67	**4.13**	33.97	**10.26**
bloat	13.79	20.01	**1.45**	34.34	**2.49**	16.40	**1.19**	20.28	**1.47**	23.57	**1.71**	115.50	**8.38**	66.24	**4.80**	220.41	**15.98**
chart	11.35	15.48	**1.36**	24.77	**2.18**	12.38	**1.09**	15.40	**1.36**	17.61	**1.55**	52.24	**4.60**	30.70	**2.70**	87.97	**7.75**
eclipse	59.44	65.88	**1.11**	72.51	**1.22**	66.98	**1.13**	139.52	**2.35**	66.28	**1.12**	166.46	**2.80**	68.12	**1.15**	268.99	**4.53**
fop	2.15	2.73	**1.27**	3.49	**1.62**	2.39	**1.11**	2.59	**1.20**	2.95	**1.37**	7.06	**3.28**	4.03	**1.87**	11.16	**5.19**
hsqldb	7.59	11.07	**1.46**	24.15	**3.18**	9.62	**1.27**	9.77	**1.29**	13.35	**1.76**	31.16	**4.11**	28.76	**3.79**	45.76	**6.03**
jython	9.15	27.87	**3.05**	30.02	**3.28**	21.57	**2.36**	22.76	**2.49**	40.59	**4.44**	81.60	**8.92**	61.25	**6.69**	127.51	**13.94**
luindex	15.94	22.41	**1.41**	25.99	**1.63**	19.16	**1.20**	19.57	**1.23**	53.20	**3.34**	91.10	**5.72**	55.12	**3.46**	97.89	**6.14**
lusearch	18.84	29.57	**1.57**	31.96	**1.70**	24.34	**1.29**	28.90	**1.53**	39.99	**2.12**	78.26	**4.15**	59.06	**3.13**	97.57	**5.18**
pmd	11.95	21.34	**1.79**	31.46	**2.63**	13.79	**1.15**	16.56	**1.39**	16.96	**1.42**	53.69	**4.49**	43.89	**3.67**	101.07	**8.46**
xalan	18.82	22.09	**1.17**	34.60	**1.84**	20.51	**1.09**	23.53	**1.25**	31.77	**1.69**	95.90	**5.10**	57.68	**3.06**	165.09	**8.77**
Geo.mean	11.09	16.64	**1.50**	23.21	**2.09**	13.96	**1.26**	16.70	**1.51**	20.95	**1.89**	56.64	**4.93**	35.44	**3.20**	85.73	**7.73**
JVM98	[s]	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**	[s]	**ovh**
compress	5.73	8.11	**1.41**	8.22	**1.43**	7.10	**1.24**	7.54	**1.32**	11.48	**2.00**	31.52	**5.50**	28.24	**4.93**	44.28	**7.73**
jess	1.46	6.15	**4.21**	9.11	**6.24**	5.39	**3.69**	6.94	**4.75**	3.91	**2.68**	12.30	**8.42**	12.31	**8.43**	21.78	**14.92**
db	14.14	14.63	**1.03**	15.94	**1.13**	14.45	**1.02**	15.25	**1.08**	15.23	**1.08**	29.92	**2.12**	14.53	**1.03**	44.98	**3.18**
javac	3.95	7.64	**1.93**	8.86	**2.24**	5.82	**1.47**	6.68	**1.69**	7.30	**1.85**	16.69	**4.23**	11.90	**3.01**	25.88	**6.55**
mpegaudio	2.47	4.31	**1.73**	5.06	**2.05**	4.39	**1.78**	5.01	**2.03**	6.24	**2.53**	14.90	**6.03**	14.84	**6.01**	21.62	**8.75**
mtrt	1.15	4.73	**4.11**	5.71	**4.97**	2.26	**1.97**	3.87	**3.37**	8.80	**7.65**	25.40	**22.09**	29.22	**25.41**	47.69	**41.45**
jack	3.48	4.94	**1.42**	7.38	**2.12**	4.02	**1.16**	4.88	**1.40**	4.71	**1.35**	11.14	**3.20**	4.94	**1.42**	16.57	**4.76**
Geo.mean	3.34	6.63	**1.98**	8.08	**2.42**	5.37	**1.61**	6.53	**1.96**	7.46	**2.24**	18.75	**5.62**	14.41	**4.32**	29.44	**8.82**

Table 2: DJProf profilers running with the AspectJ load-time weaver agent, respectively with FERRARI's dynamic instrumentation agent on a woven JDK.

setting 'AspectJ weaver' we are using AspectJ's load-time weaver (running on the original JDK), whereas the setting 'FERRARI + AJ UDI' corresponds to our framework with the AspectJ UDI (running with woven JDK). We present the execution times for the various settings and the corresponding overhead factors ('ovh'). The 'Orig.' column is the execution time for the unmodified benchmarks.

For the aspects `heap`, `lifetime`, and `cpu`, we observe rather uniform overhead. One exception is 'mtrt', where the overhead caused by the `cpu` aspect when weaving the JDK reaches a factor of 22.09, even though the aspect uses sampling to reduce the overhead. The reason for the excessive overhead is because the `cpu` aspect captures both method-execution and method-call pointcuts and invokes advices before and after each of them. This causes high overhead for applications with many invocations of short methods, as is the case for 'mtrt' [12], because of the access to the `dibFlag` on each advice invocation.

For `call-count`, the overhead is higher because this aspect captures all method calls (no sampling). We observe a high overhead for 'mtrt' in the 'AspectJ weaver' setting (factor 25.41), because the `call-count` aspect updates a global structure each time the advice is invoked. For the 'FERRARI + AJ UDI' setting, the overhead reaches a factor of 41.45. As before, this can be explained by the frequent access to the `dibFlag` within the advice, which is not present in the 'AspectJ weaver' setting.

5.4 Code Bloat Evaluation

The aspect weaver injects additional bytecodes for advice invocations. In addition, FERRARI keeps a copy of the original method bodies for the bypasses. It is therefore interesting to compare the increase of the size of the JDK due to aspect weaving respectively due to FERRARI. Table 3 shows the code bloat overhead when weaving DJProf aspects into the JDK. We observe an increase in size of 38–63% for aspect weaving, and an additional increase of 85–93% for the bypasses.

	Aspect only		Aspect + FERRARI	
Aspect	[MB]	**ovh**	[MB]	**ovh**
heap	127	**1.51**	201	**2.39**
lifetime	116	**1.38**	187	**2.23**
cpu	137	**1.63**	215	**2.56**
call-count	125	**1.49**	202	**2.40**

Table 3: Size of instrumented JDK classes. The original size is 84MB (uncompressed). 'Aspect only' is the size after weaving the profiling aspect into the JDK, and 'Aspect + FERRARI' is the final size after full instrumentation.

Even though disk space is not an important issue (each compressed JDK library is around 40MB), we need to keep a separate version of the woven JDK for each aspect. Fortunately, it is not necessary to re-instrument the JDK if interfaces between instrumented code and UDI-runtime-classes do not change. It is therefore necessary to re-weave the JDK only if advice declarations are changed, but not if only their bodies are modified. This gives some flexibility to the aspect developer for rapid prototyping, without having to re-weave the JDK for every single aspect modification.

6. RELATED WORK

The "Twin Class Hierarchy" (TCH) [13] supports user-defined instrumentation of standard Java class libraries. TCH replicates the full hierarchy of instrumented JDK classes 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 JDK version (original or instrumented). In addition, as pointed out in [27], the use of replicated classes limits the applicability of instrumentation in the presence of native code; e.g., call-backs from native code do not reach the instrumented code. Consequently, TCH is not able to safely and transparently instrument the complete JDK.

Unlike TCH, FERRARI does not use class replication techniques, but rather code duplication within the method bodies in JDK classes. FERRARI enables instrumentations with full bytecode coverage independently of the presence of native methods. Our approach ensures transparency for the application, which need not be aware whether the JDK is instrumented or not. Hence, aspect weaving into JDK classes is independent from application-level aspect weaving.

PROSE allows dynamic and runtime weaving of aspects [23]. The weaving engine resides in the JVM. PROSE introduced weaving using the Java Virtual Machine Debugger Interface (JVMDI), weaving at the just-in-time compiler level, and combining bytecode instrumentation (based on BCEL) with an extension of the Jikes RVM [1]. BEA's JRockit [3] follows a similar approach by providing AOP support directly within the JVM, therefore trading portability for performance. In contrast, our approach aims at instrumenting the whole Java class library while remaining fully portable, leveraging standard, state-of-the-art AspectJ and JVM technologies.

In [10] the authors use *J [11], a JVMPI-based tool, to gather dynamic metrics of woven programs in order to evaluate the overhead caused by AspectJ. They use annotations to keep track of the code inserted by AspectJ. Instead of resorting to a low-level tool for evaluating AspectJ, we promote using AspectJ for profiling and debugging Java programs. We currently do not consider aspects for evaluating AspectJ itself, though this could be an interesting future investigation.

Cork [16] is a tool to detect memory leaks in a time-efficient and space-efficient way. Its implementation relies on MMTk, the memory management toolkit of the Jikes RVM. In contrast, our approach is portable and JVM-independent. LeakBot [20] uses a JVMPI agent; it is designed for low overhead and uses a ranking algorithm to prune object references that have a low probability to be leaking. While our leak detection advice has not yet been optimized, its advantages are its portability and simplicity.

Aspect-based memory leak detection [19] uses aspects similar to ours but does not allow identifying memory leaks if no explicit allocation is in the application code. Our approach solves this issue thanks to aspect weaving in the JDK.

7. CONCLUSION

In this paper we presented techniques and tools that enable AOP in standard Java class libraries. Our approach relies on a generic framework that enables user-defined instrumentations with complete bytecode coverage. We adapted the popular AspectJ weaver to our framework without changing any AspectJ sources. We succeeded in removing a serious limitation of AspectJ, its inability to weave aspects into JDK classes. Our approach makes aspect-based profiling and debugging techniques practical, since it ensures a complete coverage of the Java class library. Moreover, it permits AOP for JDK developers.

We validated and evaluated our framework with existing profiling aspects; we found the execution time overhead reasonable. In addition, we presented a new aspect for memory leak detection and showed that aspect weaving in the JDK enables a more complete and accurate analysis.

An obvious limitation of our approach is the impossibility of instrumenting native code. Furthermore, static-crosscutting is currently not supported for JDK classes. Other drawbacks of our approach are code bloat and an increased overhead for load-time weaving due to FERRARI and the AspectJ adaptor.

In our ongoing research, we are developing further aspect-based profiling and debugging tools and we are working on relaxing the restrictions with respect to aspect weaving in JDK classes. In addition, we are exploring techniques for reducing the latency due to load-time instrumentation on multicores.

8. ACKNOWLEDGEMENTS

Many thanks to Jarle Hulaas for his contributions to the design of FERRARI, as well as to Jan Vitek, who has suggested using FERRARI and the AspectJ UDI for memory leak detection.

The work presented in this paper was supported by the Swiss National Science Foundation as part of the project FERRARI (project number 200021-118016/1).

9. 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] 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.

[3] BEA. JRockit. Web pages at `http://dev2dev.bea.com/jrockit/`.

[4] W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194, Tsukuba, Japan, Nov. 2005. Springer Verlag.

[5] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ 2007 (5th International Conference on Principles and Practices of Programming in Java)*, pages 135–144, Lisbon, Portugal, 2007. ACM Press.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In

OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, New York, NY, USA, Oct. 2006. ACM Press.

[7] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[8] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.

[9] 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.

[10] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 150–169, New York, NY, USA, 2004. ACM.

[11] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[12] J. Dujmovic and C. Herder. Visualization of Java workloads using ternary diagrams. *Software Engineering Notes*, 29(1):261–265, 2004.

[13] M. Factor, A. Schuster, and K. Shagin. 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*, pages 288–300, New York, NY, USA, 2004. ACM.

[14] D. Gregg, J. F. Power, and J. Waldron. A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites. *Concurrency and Computation: Practice and Experience*, 17(7–8):757–773, 2005.

[15] 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.

[16] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–38, New York, NY, USA, 2007. ACM.

[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] C. Kung and C. Ju-Bing. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007*, pages 23–28, Beijing, China, 2007. IEEE Computer Society.

[20] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP '03–Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377. Springer, 2003.

[21] ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

[22] 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.

[23] 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.

[24] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`.

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

[26] E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), USA*, volume 2487 of *LNCS*, pages 283–298, Oct. 2002.

[27] 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*, pages 89–94, New York, NY, USA, 2006. ACM.

[28] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.