Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine

Walter Binder, Alex Villazón, Danilo Ansaloni, and Philippe Moret

Faculty of Informatics, University of Lugano, Switzerland
firstname.lastname@usi.ch

Abstract. Many software engineering tools for the Java Virtual Machine that perform some form of dynamic program analysis, such as profilers or debuggers, are implemented with low-level bytecode instrumentation techniques. While program manipulation at the bytecode level is very flexible, because the possible bytecode transformations are not restricted, tool development is tedious and error-prone. Specifying bytecode instrumentations at a higher level using aspect-oriented programming (AOP) is a promising alternative in order to reduce tool development time and cost. However, prevailing AOP frameworks lack some features that are essential for certain dynamic analyses. In this paper, we focus on two common shortcomings in AOP frameworks with respect to the development of aspect-based tools – (1) the lack of mechanisms for passing data between woven advices in local variables, and (2) the support for user-defined static analyses at weaving time. We introduce @J, an annotation-based AOP language and weaver that integrates support for these two features. We illustrate the benefits of the proposed features with an example.

Keywords. Aspect-oriented programming, aspect weaving, local variables, analysis at weaving time, bytecode instrumentation, dynamic program analysis, profiling, debugging, Java Virtual Machine

1 Introduction

Bytecode instrumentation techniques are widely used for building software engineering tools that perform some dynamic program analysis, such as profilers [16, 15, 21, 8, 26], memory leak detectors [41, 27], data race detectors [11, 12], or testing tools that preserve the conditions that caused a crash [4].[1]

Java supports bytecode instrumentation using native code agents through the Java Virtual Machine Tool Interface (JVMTI) [35], as well as portable bytecode instrumentation through the `java.lang.instrument` API. Several bytecode engineering libraries have been developed, such as BCEL [36], ASM [28], Javassist [13], or Soot [37], to mention some of them.

[1] In this paper we only consider program transformations that insert bytecode, but do not alter or delete existing bytecode in a program.

However, because of the low-level nature of bytecode and of bytecode engineering libraries, the implementation of new instrumentation tools can be difficult and error-prone, often requiring high development and testing effort. For example, a frequent mistake is the incorrect update of exception handler tables, which may result in instrumented code that passes many tests, but may later fail under particular conditions. As another drawback of low-level instrumentation techniques, the resulting software engineering tools are often complex and difficult to maintain and to extend.

Aspect-oriented programming (AOP) [23] enables the specification of crosscutting 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 [30, 6, 40], which in many cases can be specified as aspects[2] in a concise manner. Hence, in a sense, AOP can be regarded as a versatile approach for specifying certain program instrumentations at a high level, hiding low-level implementation details, such as bytecode manipulation, from the programmer.

However, current AOP frameworks have not been especially designed for the implementation of instrumentation-based software engineering tools. Some important features are missing, limiting the program instrumentations that can be expressed as aspects. We found the following two features, which are not supported by prevailing AOP frameworks such as AspectJ [22], essential in various case studies where we have tried applying AOP for recasting instrumentation-based software engineering tools as aspects.

- Data passing between advices that are woven into the same method using local variables. In many instrumentation-based tools, local variables are allocated to pass data between different instrumentation sites in the code. While AspectJ's `around` advice allows passing data generated by code inserted before a join point to code inserted after a join point, it is not possible to pass data in local variables between different join points, such as from a "`before call`" advice to an "`after execution`" advice.
- Execution of custom analysis code, which only depends on static information, at weaving time. Many instrumentation-based tools perform some specific analysis to determine whether and how a particular join point shall be instrumented. For example, the listener latency profiler LiLa [21] analyses the class hierarchy to determine whether an invoked method is declared in a listener interface; only in that case the method invocation is profiled. While AspectJ lacks support for user-defined analyses at weaving time, some other AOP frameworks, such as SCoPE [3], provide such features (see Section 5 for details).

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

In order to provide these missing features in an AOP framework, we are designing @J (Aspect Tools in Java), an annotation-based aspect language and weaver, especially intended for easing the implementation of instrumentation-based software engineering tools. @J supports many AspectJ constructs (in AspectJ's annotation version, which we will call @AspectJ in this paper), and the implementation of the @J weaver reuses the code of the @AspectJ weaver as much as possible. In this paper, we focus on the new constructs offered by @J that are not available in @AspectJ.

In @J, instrumentations are expressed as *code snippets*[3] which are woven at bytecode positions specified by the snippet programmer. By default, snippets are inlined in the woven code. @J supports *invocation-local variables*, allowing snippets that are woven into the same method body to pass data in local variables [38]. @J snippets may access context information, such as static or dynamic join point instances, in the same way as in @AspectJ.

@J supports *executable snippets*, allowing the expression of custom static analyses that are executed at weaving time. An executable snippet may only access static context information. By storing values in invocation-local variables, an executable snippet can pass the results of a static analysis to other inlined snippets. Apart from writing to invocation-local variables, executable snippets must not have any side effects. An executable snippet is woven by inserting a bytecode sequence that assigns the values to invocation-local variables that the snippet has generated upon execution at weaving time. As snippets can be composed, the code inserted in a woven method at a particular bytecode position may consist of an arbitary sequence of inlined and executable snippets. Hence, custom static analyses can be embedded within inlined code snippets. For a more detailed discussion of @J features, we refer to [9].

This paper is structured as follows: Section 2 summarizes the design goals underlying @J. Section 3 discusses the distinguishing language features of @J. Section 4 gives an example @J program, illustrating the use of @J's special features. Section 5 discusses related work, and Section 6 concludes this paper.

2 Design Goals

In this section we summarize the design goals underlying @J.

- **Expressiveness:** @J is designed to allow the expression of a wide range of instrumentation-based software engineering tools. We have explored a large variety of case studies in the profiling and debugging domains in order to determine the necessary features. Examples include the dynamic metrics collector *J [16], the NetBeans Profiler [27], the latency listener profiler LiLa [21], the testing tool ReCrash [4], and the Eclipse plugin Senseo that

[3] In @J we always use the term "snippet" instead of "advice" for the code to be executed at an intercepted join point, because we found it more intuitive for programming instrumentation-based software engineering tools.

collects various dynamic metrics and runtime type information [33]. @J allows recasting the considered case studies as compact snippets that can be easily extended.
- **Efficiency:** @J shall enable the construction of efficient software engineering tools that offer the same level of runtime performance as tools programmed with low-level bytecode engineering libraries.
- **Portability and compatibility:** @J is implemented in pure Java. Snippets may be implemented in pure Java, too. Hence, snippet-based tools can be run in any standard Java Virtual Machine (JVM) (JDK 1.5 or higher). This is important, as we do not want to constrain tool users to employ a particular JVM. Snippet-based tools can be integrated into the users' preferred software development environment.
- **Full method coverage:** For many instrumentation-based dynamic analysis tools, such as profilers or memory leak detectors, it is essential that the instrumentation covers all methods executing in the JVM (which have a bytecode representation), including methods in dynamically generated or loaded classes, as well as in the Java class library. In addition, in some cases it is desirable to intercept also the execution of native methods. To ensure full method coverage, @J is based on the FERRARI framework[4] [7], which is also the basis of the MAJOR aspect weaver [39, 40].[5] Optionally, FERRARI can make use of native method prefixing, offered by the JVMTI [35] (which however requires JDK 1.6 or higher), in order to wrap native methods with bytecode versions that are amenable to snippet weaving.

3 @J Features

In this section, we firstly summarize the features of @AspectJ that are also supported in @J, and secondly explain the new features of @J that are complementary to @AspectJ.

3.1 Supported @AspectJ Features

@J supports @AspectJ pointcuts, as well as `before` and `after` advices. Static and dynamic join points are supported in the same way as in @AspectJ.

@J does not support non-singleton aspect instances using `per*` clauses (e.g., per-object or per-control flow aspect association), because @J snippets are either inlined or executed at weaving time.

@AspectJ's `around` advice is not supported in @J. The @AspectJ weaver implements the `around` advice by inserting wrapper methods in woven classes [20], which can cause problems when weaving the Java class library. For instance,

[4] `http://www.inf.usi.ch/projects/ferrari/`

[5] FERRARI prevents the execution of inserted code during JVM bootstrapping. That is, during the bootstrapping phase, @J snippets are not executed. However, full method coverage is guaranteed for the whole execution of a program's main thread, and for all threads spawned by the program.

in Sun's HotSpot JVMs there is a bug that limits the insertion of methods in `java.lang.Object`.[6] Moreover, wrapping certain methods in the Java class library breaks stack introspection in many recent JVMs, including Sun's HotSpot JVMs and IBM's J9 JVM [26, 40]; usually, there is no public documentation indicating those methods in the class library that must not be wrapped. Hence, the use of `around` advices would compromise weaving with full method coverage in many common, state-of-the-art JVMs. Nonetheless, with the aid of invocation-local variables, it is possible to emulate a common use of `around` advices as a combination of `before` and `after` advices.

Static cross-cutting (inter-type declarations) [22] enables explicit structural modifications, such as changes of the class hierarchy or insertions of new fields and methods. In contrast to AspectJ without annotations, @AspectJ restricts the possibilities of static cross-cutting. For instance, in @AspectJ, it is not possible to insert fields in existing classes.

3.2 Snippets and their Composition

While an aspect in @AspectJ starts with an `@Aspect` annotation, an @J class is annoted with `@J`, which can take some extra annotation parameters.

Snippets are public static methods with void return type annotated with `@BeforeSnippet`, `@AfterSnippet`, `@AfterReturningSnippet`, or `@AfterThrowingSnippet`. These @J annotations correspond to `@Before`, `@After`, `@AfterReturning`, respectively `@AfterThrowing` advice methods in @AspectJ, but may take some additional annotation parameters. In @J, snippets may be woven only before or after a join point; in contrast to @AspectJ, @J does not support weaving around a join point. The @J snippet annotations support the optional boolean parameter `execute` for indicating whether a snippet is to be inlined (default, `execute`=false) or executed at weaving time (`execute`=true).

In contrast to @AspectJ, snippets are always static in @J. Since snippets are inlined or executed at weaving time, it is not possible to change the snippets associated with a program at runtime. In contrast, the standard @AspectJ weaver inserts invocations of advice methods instead of inlining their bodies. The approach taken by @AspectJ has the benefit that the aspect association can be changed at runtime. However, for the purpose of @J, we consider static snippets appropriate, because snippet inlining is a prerequisite for passing data between snippets woven into the same method using local variables.

If multiple snippets match a join point, the @J programmer must specify the precedence of snippets. To this end, the @J snippet annotations support an optional integer parameter `order` (snippets with smaller `order` value come first). Weaving produces an error, if the order of multiple matching snippets is insufficiently specified.

[6] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6583051`

3.3 Invocation-local Variables

@J supports the notion of *invocation-local variables* [38], in order to allow efficient data passing in local variables between snippets. The term "invocation-local" was chosen to imply that the scope of an invocation-local variable is one invocation of a woven method. Invocation-local variables are accessed through public static fields that have `@InvocationLocal` annotations. Within snippets, invocation-local variables can be read and written as if they were static fields. For each invocation-local variable accessed in a woven method, a local variable is allocated and the bytecodes that access the corresponding static field are simply replaced with bytecodes for loading/storing from/in the local variable.

Each invocation-local variable is initialized in the beginning of a woven method with the value stored in the corresponding static field, which is assigned only during execution of the static initializer.[7] This implies that the @J class holding the snippets and the invocation-local variables is also loaded in the JVM, although the snippets are never invoked at runtime, and the static fields corresponding to invocation-local variables are assigned values only by the corresponding static initializer. As an optimizing, the initialization of local variables corresponding to invocation-local variables in woven methods is skipped, if the weaver can statically determine that the first access is a write.

3.4 Snippet Execution at Weaving Time

In many instrumentation-based software engineering tools, we found optimizations where some static analysis is performed at instrumentation time in order to decide whether and how to instrument a particular location in the bytecode. Standard AspectJ does not support the execution of custom analysis code at weaving time, making it impossible to recast such optimizations in aspects.

@J introduces *executable snippets*, which are executed at weaving time. Executable snippets produce weavable results by writing to invocation-local variables. In the woven method, a bytecode sequence is inserted that reproduces the values of the written invocation-local variables. Executable snippets may access only static context information, such as static join points, and must not have any side effects apart from writing to invocation-local variables of primitive type or of type `java.lang.String`. Executable snippets must not read any invocation-local variable.

Instead of inlining an executable snippet, the @J weaver creates an environment that enables snippet execution for matching join points at weaving time. To this end, the weaver generates a class holding the executable snippets (in a transformed version) and the invocation-local variables. The transformed snippets are instrumented so as to provide the set of written invocation-local variables upon completion. The resulting class is loaded at weaving time, and for each matching join point, the corresponding transformed snippet method is called, passing

[7] The Java memory model [19, 25, 18] ensures that the value assigned by the static initializer is visible to all threads.

the needed static context information of the join point as arguments. Note that this may require allocating static join point instances at weaving time, if an executable snippet makes use of it. After snippet execution, the weaver inlines a bytecode sequence in the woven method that assigns the written invocation-local variables with the respective constants (which are added to the constant pool of the class holding the woven method).

A typical use of an executable snippet is to run some static analysis at weaving time, producing a boolean value in an invocation-local variable indicating whether the join point shall be instrumented. The executable snippet is composed with a normal (inlined) snippet, which is a conditional statement on the value of the boolean invocation-local variable. Evidently, in case the condition is false, the inlined snippet is dead code, which is likely to be eliminated by the just-in-time compiler of the JVM. @J does not perform any bytecode optimization, such as dead code elimination, since we assume that the snippet-based software engineering tools will execute on standard, state-of-the-art JVMs, which already include sophisticated optimizations upon bytecode compilation. A detailed example involving an executable snippet is presented below.

4 Case Study: Recasting LiLa

In this section, we discuss an example @J program that recasts the listener latency profiler LiLa [21].

Listener latency profiling helps developers locate slow operations in interactive applications, where the perceived performance is directly related to the response time of event listeners. LiLa[8] is an implementation of listener latency profiling based on ASM [28], a low-level bytecode engineering library.

The response time for handling an event relates to the execution time of an invoked method on an instance of a class implementing the `java.util.EventListener` interface. In order to reduce profiling overhead, LiLa does not instrument all methods in each subtype of `java.util.EventListener`, but restricts the instrumentation to those methods that are declared in an interface. Hence, LiLa analyzes the class hierarchy to determine which methods to instrument. This optimization reduces profiling overhead at runtime, because less methods are instrumented.

Even though it is possible to recast the basic profiling functionality of LiLa as an aspect in AspectJ, for example, using the `around` advice to measure response time by surrounding the execution of every event-related method, the optimization that reduces the number of instrumented methods cannot be performed at weaving time.

In Figure 1, we show how the static analysis at weaving time is implemented in @J with the executable snippet `analyzeNeedsProfiling(...)`. The result of the snippet execution at weaving time is stored in the invocation-local variable `needsProf`. The `takeStartTime()` snippet, which is inlined after the bytecodes

[8] `http://www.inf.usi.ch/phd/jovic/MilanJovic/Lila/Welcome.html`

that result from executing `analyzeNeedsProfiling(...)`, records the starting time only if the static analysis determined that profiling was needed. The invocation-local variable `start` stores the starting time for later use in the same woven method. The `takeEndTimeAndProfile(...)` snippet intercepts (both normal and abnormal) method completion. The listener object is made accessible within the body of the snippet through the expression "`this(listener)`" in the pointcut declaration. Whenever the execution time exceeds the given threshold, the method `profileEvent(...)` (not shown in the figure) logs an identifier of the intercepted method (conveyed by the static join point), the target object, and the execution time. This information helps developers locate the causes of potential performance problems due to slow event handling.

The example in Figure 2 illustrates the weaving of an `EventListener` implementation. For the sake of easy readability, we show the transformations conceptually at the Java level, whereas the @J weaver operates at the bytecode level. The method `actionPerformed(ActionEvent)` is declared in the implemented interface and needs to be profiled, whereas the method `notDeclaredInInterface()` does not require profiling. Figure 2(b) shows the result of weaving. The interesting part is how the result of the static analysis is stored in the invocation-local variable `needsProf`. The woven code is quite long, since there is a significant amount of dead code. A state-of-the-art compiler will detect and eliminate the dead code, yielding the optimized code shown in Figure 2(c).

5 Related Work

The AspectBench Compiler (*abc*) [5] eases the extension of AspectJ with new pointcuts [1, 11, 14]. Even though the new pointcuts of @J could be implemented as an extension using *abc*, we opted for an annotation-based snippet development style in order to rapidly prototype @J features and therefore focus on the weaving part, rather than on the aspect language front-end. In addition, adapting the @AspectJ weaver to use FERRARI [7] for full method coverage turned out to cause less development effort than modifying *abc*.

Nu [17] enables extensions using an intermediate language model and explicit join points [32]. *Nu* adopts a fine-grained join point. Similar to @J, it allows to express aspect-oriented constructs in a flexible manner. While *Nu* is based on a customized JVM, @J is compatible with standard JVMs and uses standard Java compilers.

Steamloom [10] provides AOP support at the JVM level, which results in efficient runtime weaving. Steamloom enables the dynamic modification and reinstallation of method bytecodes and provides dedicated support for managing aspects. Steamloom uses its own aspect language and provides a parser to support AspectJ-like pointcuts. Steamloom is based on the Jikes RVM [2] and supports thread-locally deployed aspects. In order to support thread safety, Steamloom uses code snippets that are inserted before every call to advices, so as to verify whether the advice invocation for the current thread should be active

```
@J
public class LiLa {
  // listeners executing less than 100 ms (100,000,000 ns) are not logged
  public static final long THRESHOLD_NS = 100L * 1000L * 1000L;

  @InvocationLocal
  public static long start; // stores starting time of listener execution

  @InvocationLocal
  public static boolean needsProf; // stores result of static analysis

  // pointcut matching the execution of any method
  // in any subtype of the EventListener interface
  @Pointcut( "execution(* java.util.EventListener+.*(..))" )
  void listenerExec() {}

  // static analysis at weaving time (result stored in invocation-local variable);
  // jpsp provides method details (package, class, name, signature)
  @BeforeSnippet( pointcut = "listenerExec";
                  execute = true;
                  order = 1; )
  public static void analyzeNeedsProfiling(JoinPoint.StaticPart jpsp) {
    needsProf = isInterfaceMethod(jpsp); // not shown here
  }

  // store starting time upon listener entry, if the static analysis
  // considers profiling necessary
  @BeforeSnippet( pointcut = "listenerExec";
                  order = 2; )
  public static void takeStartTime() {
    if (needsProf) start = System.nanoTime();
  }

  // profile listener execution upon completion, if the static analysis
  // considers profiling necessary and the execution time exceeds the threshold
  @AfterSnippet( pointcut = "listenerExec && this(listener)"; )
  public static void takeEndTimeAndProfile(JoinPoint.StaticPart jpsp,
                                           java.util.EventListener listener) {
    if (needsProf) {
      long exectime = System.nanoTime() - start;
      if (exectime >= THRESHOLD_NS)
        profileEvent(jpsp, listener, exectime); // not shown here
    }
  }
  ...
}
```

Fig. 1. Listener latency profiler LiLa expressed in @J

or not. Similar to Steamloom, PROSE [31] also provides aspect support within the JVM. PROSE combines bytecode instrumentation and aspect support at the just-in-time compiler level with an extension of the Jikes RVM. Unfortunately, these approaches require a customized JVM, thus limiting extensibility and portability.

Prevailing AspectJ weavers do not support the execution of custom analysis code at weaving time, which typically only depends on static information. SCoPE [3] is an AspectJ extension that partially solves this problem by allowing analysis-based conditional pointcuts. Similarly, the approach described in [24]

(a) Before weaving:

```
class ExampleListener implements ActionListener {
  public void actionPerformed(ActionEvent e) { doSomething(); }
  public void notDeclaredInInterface() { doSomethingElse(); }
  ...
}
```

(b) Woven code:

```
class ExampleListener implements ActionListener {
  private static final JoinPoint.StaticPart
    jpsp1 = ..., // representing actionPerformed
    jpsp2 = ...; // representing notDeclaredInInterface

  public void actionPerformed(ActionEvent e) {
    long start = 0L;
    boolean needsProf = true;
    if (needsProf) start = System.nanoTime();
    try { doSomething(); }
    finally {
      if (needsProf) {
        long exectime = System.nanoTime() - start;
        if (exectime >= LiLa.THRESHOLD_NS)
          LiLa.profileEvent(jpsp1, this, exectime);
      }
    }
  }

  public void notDeclaredInInterface() {
    long start = 0L;
    boolean needsProf = false;
    if (needsProf) start = System.nanoTime();
    try { doSomethingElse(); }
    finally {
      if (needsProf) {
        long exectime = System.nanoTime() - start;
        if (exectime >= LiLa.THRESHOLD_NS)
          LiLa.profileEvent(jpsp2, this, exectime);
      }
    }
  }
  ...
}
```

(c) Optimized code (e.g., by a just-in-time compiler that eliminates dead code):

```
class ExampleListener implements ActionListener {
  private static final JoinPoint.StaticPart
    jpsp1 = ..., // representing actionPerformed
    jpsp2 = ...; // representing notDeclaredInInterface

  public void actionPerformed(ActionEvent e) {
    long start = System.nanoTime();
    try { doSomething(); }
    finally {
      long exectime = System.nanoTime() - start;
      if (exectime >= LiLa.THRESHOLD_NS)
        LiLa.profileEvent(jpsp1, this, exectime);
    }
  }

  public void notDeclaredInInterface() { doSomethingElse(); }
  ...
}
```

Fig. 2. Weaving and optimization of an example EventListener implementation

enables customized pointcuts that are partially evaluated at weaving time. @J supports custom analysis through snippets that are executed during the weaving.

Maxine [34] is a meta-circular research VM implemented in Java. Maxine uses a layered compiler with different intermediate representations. Instead of writing the code in a particular intermediate representation to add a runtime feature, Maxine allows developers to write snippets directly in Java, which are compiled into the corresponding intermediate representation. This approach decouples runtime features from compiler work. The Ovm [29] virtual machine follows a similar approach, where a high-level intermediate representation eases the customization for building language runtime systems, so as to define new operations and to modify the semantics of existing ones.

6 Conclusion

Low-level bytecode instrumentation is a prevailing technique for implementing tools that perform some kind of dynamic program analysis, such as profiling. As implementing instrumentations at the bytecode level is tedious and error-prone, specifying dynamic analysis tools with high-level AOP is a promising approach for reducing tool development costs, for improving maintainability, and for easing extension of the tools.

Unfortunately, many prevailing AOP frameworks, such as AspectJ, lack certain features that are important for developing efficient dynamic analysis tools for certain purposes. We identified two missing features in AspectJ that we consider essential for tool development: efficient data passing between woven advices in local variables, and the execution of custom static analyses at weaving time.

In this paper, we propose the annotation-based AOP framework @J, which is based on @AspectJ and incorporates support for these two features. As examples, we recast an existing tool based on low-level bytecode manipulation as an @J program, illustrating the use of @J's distinguishing features. The resulting tool is compactly implemented within a few lines of code.

Regarding limitations, @J suffers from the same problem as any other framework relying on Java bytecode instrumentation. The JVM imposes strict limits on certain parts of a class file (e.g., the method size is limited); these limits may be exceeded by the code inserted upon aspect weaving. Our approach aggravates this issue by inlining snippets, usually increasing the code bloat. Nonetheless, we have not yet encountered any problems due to code growth in practice.

7 Acknowledgements

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

References

1. S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for AspectJ. In *ACP4IS '09: Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 43–48, New York, NY, USA, 2009. ACM.
2. 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.
3. T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, 2007. ACM.
4. S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In J. Vitek, editor, *ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565, Paphos, Cyprus, 2008. Springer-Verlag.
5. 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.
6. 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.
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. W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J - Towards Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine. In *VMIL '09: Proceedings of the 3th Workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms*, New York, NY, USA, 2009. ACM.
10. C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 109–124, New York, NY, USA, 2006. ACM.
11. E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008*, pages 155–165, New York, NY, USA, 07 2008. ACM.
12. F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 221–230, New York, NY, USA, 2008. ACM.

13. 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.
14. B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 60–71, New York, NY, USA, 2008. ACM.
15. 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.
16. 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.
17. R. Dyer and H. Rajan. Nu: A dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, pages 191–202, New York, NY, USA, 2008. ACM.
18. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
19. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.
20. 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.
21. M. Jovic and M. Hauswirth. Measuring the performance of interactive applications with listener latency profiling. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 137–146, New York, NY, USA, 2008. ACM.
22. 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.
23. 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.
24. K. Klose, K. Ostermann, and M. Leuschel. Partial evaluation of pointcuts. In *PADL*, pages 320–334, 2007.
25. 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.
26. 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.
27. NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`.
28. ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

29. K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 67–76, New York, NY, USA, 2003. ACM.
30. 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.
31. 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.
32. H. Rajan. A Case for Explicit Join Point Models for Aspect-Oriented Intermediate Languages. In *VMIL '07: Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*, page 4, New York, NY, USA, 2007. ACM.
33. D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting Static Source Views in IDEs with Dynamic Metrics. In *ICSM '09: Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pages 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.
34. Sun Microsystems, Inc. The Maxine Virtual Machine. Web pages at `http://research.sun.com/projects/maxine/`.
35. Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.
36. The Apache Jakarta Project. The Byte Code Engineering Library (BCEL). Web pages at `http://jakarta.apache.org/bcel/`.
37. 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.
38. A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Advanced Dynamic Runtime Adaptation for Java. In *GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*. ACM, Oct. 2009.
39. 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.
40. 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.
41. G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 151–160, New York, NY, USA, 2008. ACM.