Envelope-Based Weaving for Faster Aspect Compilers

Christoph Bockisch   Michael Haupt   Mira Mezini   Ralf Mitschke
Software Technology Group
Department of Computer Science
Darmstadt University of Technology, Germany
{bockisch,haupt,mezini}@informatik.tu-darmstadt.de
mitschke@rbg.informatik.tu-darmstadt.de

Abstract: Current compilers for aspect-oriented programs are rather costly in terms of memory and time consumed at compile-time. This is because member accesses that are advised by aspects, generally, occur several times in the program code. These sites are in complex methods whose control flows must be updated when weaving advice. In this paper we present a new weaving technique that introduces indirections into the program code which makes weaving advice easier. We will show that the time and memory consumption at compile-time benefits from this approach and still the loss of runtime-performance is acceptable as the introduced indirections are optimized away by the execution environment.

1 Introduction

Aspect-oriented programming (AOP) [K^{+}] is a paradigm for modularizing crosscutting concerns. The most prominent kind of AOP languages [Ker] are AspectJ-like languages [Aspa]. The fundamental concepts of the AspectJ flavor of AOP are *join points*, *pointcuts* and *advice*. A pointcut is a query that quantifies over join points, which are points in the *execution* of the program. Thereby a set of related join points is defined. Pointcuts are composed of primitive queries which are called *pointcut designators* (PCDs). Advice are pieces of functionality that can be attached to pointcuts, taking semantic effect when the respective pointcuts “match”, i. e., when join points referred to by the pointcut are reached. Examples of join points in AspectJ’s model are method/constructor executions/calls, field accesses, initializations etc.

Join point shadows (JPS) map dynamic join points to their corresponding static code structures (expression, statement or block in the AspectJ language) that *might* yield dynamic join points during execution [MKC]. The shadows of join points matched by pointcuts are calculated given a pointcut and the kind of advice, i.e., `before` or `after`.

In current compiler technology for AspectJ-like languages, JPSs play a twofold role: (a) tools like the AspectJ Development Tools Eclipse plugin [CCHW05] use them to make the crosscutting structure explicit in the code, and (b) the weaver uses them as the points at which it weaves code for dispatching to advice. We argue that using join point shadows as weaving locations has some problems w.r.t. the compilation efficiency. Shadows for pointcuts that match field access and method and/or constructor call join points can in

general be spread all over the program; even worse, they may appear several times. Hence, the same weaving action must be executed repeatedly at each of these.

For illustration, consider the pseudo-code in Listing 1, consisting of two classes `A` and `B` and the aspect `Log`. The latter executes advice at any point during the execution, when fields of the class `A` are accessed[1]. The shadows of `Log`'s pointcut are marked by an asterisk (#) in Listing 1.

```
1  class A {
2   int x; }
3
4  class B {
5   void o() {
6     if (...) .. (#) a.x..;
7     ..(#) a.x..
8   }
9   void p() { .. (#) a.x .. } }
10
11 aspect Log {
12  before(): get(* A.*) {..} }
```

Listing 1: Weaving locations with current compilation techniques.

As performance evaluations indicate, this causes significant overhead. The reason for which is the complexity of a weaving action. First, when weaving advice dispatch code at a join point shadow, care must be taken of the control flow of the method whose instructions are affected. At bytecode level, control flow is realized by jump instructions with relative offsets, which must be updated to skip the additional instructions that comprise the advice dispatch logic. There are other constructs in Java bytecode [LY99] such as exception tables or debug information that also must be kept up-to-date.

Second, in order to cope with the complexity of the bytecode instrumentation, bytecode toolkits that abstract from details of the Java bytecode and collections of meta-data are generally used. The AspectJ compiler uses the BCEL bytecode manipulation toolkit [BCE]; the AspectBench Compiler (`abc`) [A$^+$05] employs Soot [Soo]. The use of bytecode toolkits and collections of meta-data, however, increases the compilation time and the memory usage significantly.

Decreasing the compile-time of AspectJ programs is a relevant issue. As reported in [HH], the time needed by the AspectJ compiler increases considerably, compared to the time needed by a conventinal compiler even to compile programs without aspects. Poor compile time performance impacts the efficiency of the development process since it slows down the "compile-test-debug" cycle. Furthermore, efficient compilation techniques are crucial for systems that support runtime weaving [PRO, Ste, Aspb].

In this paper, we introduce a new technique for weaving AspectJ-like aspects which decreases both the number of places where advice dispatch logic must be woven in and the complexity of a weaving action. We present the concept behind the approach and issues to be considered, discuss the current implementation, its limitations and ways to address them. We evaluate the approach and show that it indeed speeds up compilation.

The remainder of this paper is organized as follows. Sec. 2 introduces the concept of our approach. Sec. 3 presents the implementation details. Sec. 4 discusses other weaving approaches, against which the approach proposed here is compared by means of empirical

[1] It could have been defined to match calls to methods of `A`, as well. The discussion is not affected.

evaluation in Sec. 5. Sec. 6 summarizes the paper as well as outlines open issues and possible solutions to them.

2 Envelope-Based Weaving in a Nutshell

To decrease the number of weaving locations and, hence, the amount of weaving required, we introduce accessors (getters and setters) for fields and proxies for methods. We will use the term *envelope methods* (envelope for short) to uniformly refer to accessors and proxies. Envelope methods are generated for any method and field in the declaring class. Envelopes simply call the original method, respectively perform the original field access, and then return. Furthermore, method call/field access sites in the program are transformed such that instead of directly calling a method or accessing a field, they call the respective envelope methods.

For illustrating the transformation process, Listing 2 shows a transformed version of the code in Listing 1. Consider the class `A` in Listing 2 and compare it to its counterpart in Listing 1. The version of `A` in Listing 2 has two new accessors for `x`, `getX` and `setX`. Also, the code of `B` has been transformed to replace any direct access to `x` by a call to the respective envelope. For reasons of brevity and simplicity, no envelopes for the method `B.o()` and `B.p()` are shown in Listing 2.

```
1  class A {
2    int x;
3    int getX() { (#) return x; }
4    void setX(int value){x = value;}}
5
6  class B {
7    void o() {
8      if(..) ..a.getX()..;
9      ..a.getX()..
10   }
11   void p() { ..a.getX().. } }
12
13 aspect Log {
14   before() : get(* A.*) {..} }
```

Listing 2: Weaving locations in the envelope-based weaving style.

The effect of the code generation and transformation just explained is threefold. First, it simplifies the search for weaving locations. Method call/execution and field access join points can only occur within envelopes, i.e., envelopes are the only weaving locations and the search for weaving locations can be directly targeted to them. Furthermore, the number of envelopes per member is limited: There is only one envelope per field and kind of access; the number of envelopes per method is equal to the number of overridden versions of the method.

For example, there is exactly one weaving location (#) for the pointcut in Listing 2. It is located in the respective getter in the class declaring the field. A conventional weaver must consider every *get* bytecode instruction as a potential weaving location. In our particular example, weaving location search could be even further optimized: only accesses to fields of class `A` are selected by the pointcut, so only envelopes within `A` are relevant to the search; a similar restriction of the search area is not possible with a conventional weaver.

Second, and more importantly, with the envelope-based weaving the number of advice weaving actions is reduced. Advice is woven only once per affected field. In the example,

only one member matches the pattern in the pointcut; there is only one weaving location, i. e., the weaving process has to be performed only once, rather than three times as with conventional weaving. As far as method calls are concerned, the hypothesis underlying our approach is that, in general, there are much more call sites than declarations of methods. An analysis of the SpecJVM98 [SPE] benchmark, a representative set of client applications, showed that the hypothesis is correct: For the complete benchmark suite, there are about three times as many calls to application methods than declared application methods. In cases when there are less call sites for a method than polymorphic implementations of the method, our approach has to weave in more places than AspectJ. But, as explained, this is not common place.

Finally, the most important effect of introducing envelopes is that the weaving process itself becomes extremely simple. Envelope methods have a very primitive sequential structure free of control statements (e. g., `if` or `while`), exception handlers, or debugging information. This means that all precautions with regard to maintaining the control structure of methods affected by weaving can be dropped. Weaving becomes simple enough to be performed by modifying the Java bytecode directly; no tools for complex bytecode manipulation and less meta-data are needed.

There are some issues with our weaving approach, that ought to be addressed. First, envelopes add an indirection, affecting the runtime behavior of the compiled program. However, due to modern just-in-time (JIT) compiling virtual machines [Jik,Hot], the introduced indirection only has small impact on the runtime performance, as our evaluation will also confirm. A JVM using a JIT compiler compiles Java bytecode to machine code and natively executes the machine code. At this step, several optimizations are applied to generate high-performance machine code, one of which is inlining [B+99]. The JIT compiler may decide to inline a method instead of calling it via a dispatch table when the call at hand is non-polymorphic. Small methods are more likely to be inlined because the ratio of additional time for the JIT compilation and the time saved by cheaper dispatch is better than for larger methods. Our envelope generation makes use of this knowledge: the generated code embodies hints for the JIT enabling the latter to optimize away the introduced indirections.

The above said, we would like to add the following remark. As far as systems with static weaving are concerned, we view our approach as complementary to conventional weaving techniques. We envisage it to be used for compiling the program at development time when a fast "compile-test-debug" cycle is important; compiler techniques that aim at runtime-optimal code can than be used for production builds. We also see our approach as a promising technique for aspect-oriented environments with runtime weaving. In those systems, poor compile-time is also poor runtime, because deploying an aspect imposes a delay on the running application.

Another issue with our approach is that envelopes operate on the callee site and, hence, do not have direct access to the caller object. To support the `this` pointcut designator of AspectJ, we provide an extension to standard Java 5 Virtual Machines that makes this kind of context accessible again. The extension is implemented by means of the Java Native Interface (JNI) [JNI] and the JVM Tools Interface (JVMTI) [JVM]. As we will see in section 5 the runtime penalty imposed by the extension is negligible.

Finally, unlike conventional weaving approaches, envelope-based weaving distinguishes between weaving locations and join point shadows. Our weaving locations do not directly correspod to the crosscutting structure expressed by the pointcut and can not be used to display feedback to the programmer in an integrated development environment. We will discuss in section 6 how this feedback can still be provided by our approach without significantly slowing down performance.

3 Implementation Issues

In this section we present the technical details of the approach. We start by short presentations of the programming model and of background information on the Java classfile format.

3.1 Programming Model

We have implemented the envelope-based weaving approach for a subset of AspectJ's pointcut designators[2], namely call, get, and set and the dynamic PCDs this and target. As we will discuss, the approach applies to other kinds of join points, as well. In the current version, we only support before and after advice but no around advice which we will address in future work.

Since our focus is on the weaving technique, we do not provide a dedicated source-code compiler for aspects. Rather, advice functionality must be implemented as normal virtual methods in standard Java classes, which are compiled by a Java source-code compiler. The class must have a public static method aspectOf(), which returns an instance of the class[3]. An aspect is specified in a file as a list of pointcut-and-advice definitions in the following format:

```
before(<parameters>) | after(<parameters>) :
  <pointcut> :
  <aspect class>.<advice method>;
```

Up to the second colon, this is the same syntax as for pointcut-and-advice in AspectJ. <parameters> specifies that values from the context of a join point are passed to the advice[4]. The following example shows how to define a before advice for all calls to public methods. The advice functionality is defined in the method callAdvice() in the class MethodInvocationAspect.

```
before() : call(public * *.*(..)) : MethodInvocationAspect.callAdvice;
```

[2]The implementation can be downloaded at [Env].
[3]This is in the vein of how the AspectJ compiler generates Java bytecode.
[4]The advice method's signature must declare the same parameter list.

3.2 The Java Class File Format

Since we directly modify Java bytecode class files [LY99], a short explanation of their elements that are affected by the implementation is in place here. A class file consists of several sections, of which only the following are relevant for our discussion: type information, constant pool, fields, and methods. The constant pool contains, among other values, the fully qualified names of classes and members that are referenced from the class' code.

A method declaration consists of constant pool references that resolve to the method's name and signature (the list of the method's parameters and the return type) and the access flags. Access flags reflect the modifiers that are declared for the method in the source code. Methods that are neither native nor abstract, have a `Code` attribute[5] that stores the method body. Apart from the instructions, the `Code` attribute contains several values that must be considered. Besides the `code length` value which must always contain the correct size, the values `max stack` and `max locals` must be taken care of. They specify the maximum depth of the operand stack and the number of words needed to store local variables.

3.3 Introducing Envelopes

Envelopes are generated in the same class in which the enveloped member is defined and have the same visibility as their enveloped members.

Method envelopes. Any method to be enveloped is made private and its signature is changed by adding an additional parameter, the type of which serves the purpose of giving the method to envelope a new identity. We basically create a new overloaded method this way[6].

The generated envelope gets the name and the signature of the original method. The body of the generated envelope method performs three tasks: (1) push the arguments for the actual method call onto the operand stack, (2) call the original implementation, (3) return to the caller. The arguments to be passed to the original method are those passed to the envelope method. As the signature of the original method was extended by one object, the envelope additionally passes the constant `null` to the enveloped method.

The call from the envelope to the enveloped method is dispatched statically because the enveloped method is made private. This gives the JIT compiler a hint that the called method can be inlined, which basically optimizes away the envelope.

Since envelope methods get the name of the original method, call sites do not need to be modified.

[5]Other attributes are irrelevant for our discussion; they are neither read nor modified by our implementation.

[6]We change the method's identity by changing its signature rather than its name, because renaming would violate some presumptions made by the bytecode verifier.

Field envelopes. The naming scheme for field envelopes encodes the name of the declaring class, the field name, and the kind of access. The inclusion of the class name ensures that no envelope method with the same name is accidentally inherited from a superclass. Field envelopes cannot be inherited because they are generated with the modifier final.

The body of a field envelope method has a similar form as a method envelope's body. First the context for the field access is established. This is composed of the object whose field is to be accessed (`this` in the context of the envelope) and, in the case of a field-write access, the field's new value. Next, there are an instruction for direct field access and an appropriate returning instruction.

To replace direct field accesses with calls to respective field envelopes, the code of all methods that access fields is modified. All instructions for direct field access are replaced by instructions calling the respective generated envelope methods. The name of the envelope method to use for the call is derived from the field reference and the kind of access. The instructions for direct field access can simply be overwritten with the instructions that call the envelope methods because they occupy the same number of bytes.

Envelope methods for fields are quasi-statically dispatched as they have no super implementation and are final. Even though they are invoked like virtual methods, they behave as if they were being statically dispatched and the JIT compiler can recognize and exploit this fact to generate efficient machine code.

Special cases, for which no envelopes are generated. In two cases no envelopes are generated. First, for fields defined in interfaces; defining non-abstract methods in interfaces is not possible[7]. A workaround is to generate a dedicated class containing envelopes for interface fields. Second, no envelopes are generated for native methods because this would require that the signature of the native implementation is changed[8].

3.4 Weaving in the envelopes

Weaving advice associated with static pointcuts (`call`, `get`, or `set`) is simple. Envelopes always have the same structure (schematically shown by the graphic on the left-hand side of Fig. 1) and there is exactly one location within their code where to insert the generated advice dispatch logic. Depending on the kind of advice, this is either right before the first instruction of the envelope or right after the instruction in the middle of the envelope, where the enveloped member is accessed. To facilitate weaving, auxiliary data structures are generated during the transformation phase. A byte array that holds the bytecode for envelopes is stored for each possible join point; the index of the instruction where the enveloped member is accessed is also marked, so that the position can be accessed directly if needed to weave an after advice.

[7] A field access is replaced by a call to the respective envelope, only if the field is not defined in an interface.

[8] The native implementation is looked up using a naming scheme. For the native overloaded method `void m(int)` in class `pckge.A`, there must be a function named `Java_pckge_A_m__I` in a native library [JNI].

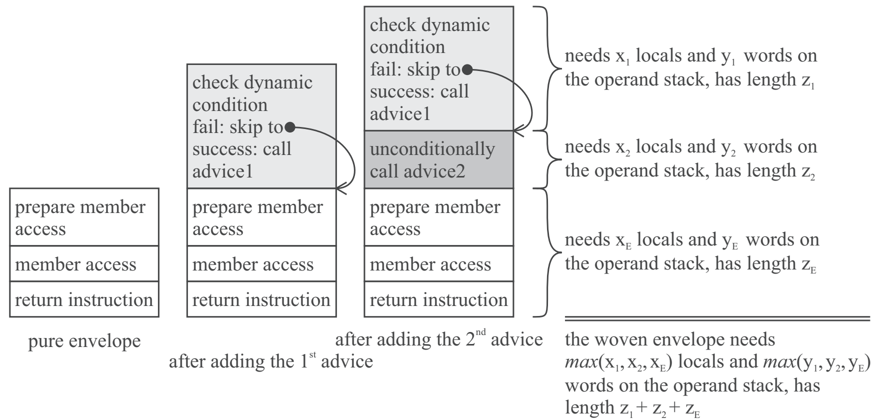

Figure 1: How blocks of advice dispatch code are woven.

The advice dispatch block is also simple. For any pointcut-advice `before() / after() : <pointcut> : SomeAspect.someAdvice`, the code to weave in corresponds to the source code `SomeAspect.aspectOf().someAdvice()`. Pointcuts can also extract values from the context of matched join points to be passed to advice. In this case, additional instructions are generated within the advice dispatch logic that pass the respective values to the invoked advice method.

Weaving for dynamic pointcut designators (`cflow`, `this`, `target`, `within`) is slightly more complicated. In their presence advice is only *possibly* executed at a weaving location. Conditional logic (called residue in [HH]) must precede the advice invocation in the advice dispatch logic, together with a branch instruction that skips the advice invocation, if the check fails. The relative offset of the branch instruction is fix: the length of the following instructions for the advice dispatch. That is, a block of instructions to weave for a matching pointcut-advice contains only jump instructions targeted just behind the block, if any. Consequently, each such block can be handled separately and inserting one of them does not require to update others already inserted, as schematically shown in Fig. 1.

For the dynamic PCDs, `cflow`, `within`, and `this`, the dynamic context to be checked is not directly accessible in our weaving approach. `within` and `this` refer to context that can be resolved statically with AspectJ's weaving style. This is not true for our approach, since the advice dispatch is woven in callee's rather than in caller's context. We resolve the context exposed by these PCDs dynamically. When the JPS is executed in an envelope, both PCDs refers to the method on the call stack just below the envelope. This context is provided in our approach by means of a static native method that uses the JVM Tools Interface (JVMTI). Right now, we do not support `cflow`; we plan to implement it in a similar way as the AspectJ compiler [HH].

After weaving the dispatch logic for all applicable pointcut-advice, an envelope's code length, its number of local variables, and the maximum depth of the operand stack are

adjusted. As shown in Fig. 1[9], each block of advice dispatch (with or without residues) is be regarded separately; the calculated values are aggregated. No complex analysis of the envelope method after weaving is necessary.

3.5 Weaving Location Search

Since join points only occur in our envelope methods, we can employ a different matching strategy than the AspectJ compiler. There, every possible weaving location is matched against the declared pointcuts [HH]. In contrast, we match the pointcuts only against all envelopes. In addition, the envelopes are contained in a hierarchical structure reflecting the application's class hierarchy. For pointcuts containing type restrictions, such as `call(* ClassType.*(..))`, this allows us to rule out a large number of weaving locations very fast, since envelopes not contained in `ClassType` do not have to be investigated.

3.6 Dynamic weaving

We have implemented a prototype for a dynamic weaving environment using our envelope-based weaving. This prototype is realized as a Java Agent using standard Java 5 features. The agent is called-back from the JVM to transform a Java class file just before the class is defined by the JVM. This is when the envelopes are introduced; the same representation of the transformed class file as presented in Sec. 3.4 is kept in memory. When an aspect is deployed, weaving location search and weaving is performed as described in Sec. 3.5, respectively 3.4. Classes that contain envelopes which have been modified in the weaving step are redefined using the HotSwap [Dmi01] technology.

4 Related Work

In this section, we discuss related work against which we will evaluate envelope-based weaving in the following section.

4.1 Aspect Compilers

There are two compilers publicly available for AspectJ-like AOP languages: the `ajc` compiler included in the AspectJ distribution [Aspa], and the AspectBench Compiler [A$^+$05] (`abc` for short).

[9] The precedence of different advice that affect the same weaving location has not yet been addressed and is arbitrary in the current implementation.

The AspectJ compiler employs some optimizations targeted at reducing the compilation time [HH]. Nevertheless, even in the absence of aspects, the compilation with `ajc`, version 1.1, is about 62 % slower than with `javac`, version 1.4. This is because the weaver scans all binaries produced by the source code compiler for weaving instructions.

The focus of `abc` is on extensibility. It is designed to facilitate easy implementation of new language features and optimizations [A+05]. The customization power is derived from Polyglot [Pol], a compiler framework, and Soot [Soo] on which `abc` is built. In some cases, `abc` generates code that exhibits better run-time performance as compared to `ajc` [A+b]. However, at the cost of a compilation time that is slower than with `ajc` [A+a]. For this reason, we will not further consider `abc` in the evaluation section.

4.2 Runtime Weaving

In the following, we discuss three AOP implementations for Java offering runtime weaving with regard to how they weave aspect functionality into base application code. Each of them represents a specific approach to runtime weaving.

AspectWerkz 2.0 [Aspb] transforms possible join point shadows through a compiler or at load-time. Advice invocations are either woven in during the transformation step or the shadows are merely *prepared* for later attachment of advice. Several other AOP implementations with runtime weaving follow basically the same approach, like JAC [PSDF] and JBoss AOP [JBo].

At first sight, AspectWerkz's weaving bears some similarity with our approach. In both cases generated methods are invoked at join point shadows. However, unlike our approach, AspectWerkz does not aim at reducing the number of weaving locations. The purpose of generated methods rather is to facilitate late introduction of advice. Wrappers are generated on the caller site: One wrapper per affected member and per method in which the access occurs. Similar to conventional weaving with `ajc`, the same weaving action must be executed multiple times. Also, unlike our approach, values from the original join point shadow context are passed to the invoked generated method, which requires costly modifications of the method containing the join point shadow.

Steamloom [BHMO, H+, Ste] provides support for aspect-oriented mechanisms at the execution layer. It is implemented as an extension to IBM's Jikes Research Virtual Machine (RVM) [Jik]. In Steamloom, weaving location retrieval and weaving are entirely done at runtime. When aspects are woven into a running application, advice method invocations are inserted at all corresponding weaving locations, and the affected methods are recompiled with the JIT compiler.

Woven code is minimal in Steamloom. Only instructions necessary to prepare and execute advice invocations are woven in at weaving locations which are the same as join point shadows, similar `ajc`. Therefore, unlike in the approach presented here, no indirections

are introduced. The overhead induced by recompiling methods is comparable to the overhead of redefining classes through HotSwap. Nevertheless, Steamloom performs more expensive queries on the application's bytecode than our envelope-based weaving approach, when evaluating pointcuts. Its weaving approach is basically that of the AspectJ compiler, except that it is performed at the VM level.

PROSE [PGA, PAG, PRO], the third approach we take into account, exists in two versions. The one we consider here (called PROSE 1) relies on a standard JVM's debugger interface and is unique in that it does not instrument an application's code at all. Upon weaving an aspect into a running application, PROSE 1 registers breakpoints at the JSPs of the aspect. To that end, a certain amount of querying the loaded classes' bytecodes is necessary. Once a breakpoint is registered and reached, execution branches to the PROSE infrastructure, which looks up the appropriate advice functionality and invokes it. In PROSE 1, no methods have ever to be recompiled, which is an advantage over the other discussed approaches, and over our envelope-based weaving implementation. The downside of its approach is twofold. First, it needs time to query the application bytecodes to identify breakpoints. Second, context switches at debugger breakpoints are very expensive.

The second version of PROSE basically pursues the same strategy for deploying aspects. However, it does not rely on the JVM's debugger interface but provides similar features by means of an extension of the Jikes RVM.

5 Evaluation

In this section, we compare envelope-based weaving with AspectJ 1.2.1 [Aspa], AspectWerkz 2.0 [Aspb], PROSE 1.2.1 [PRO], and Steamloom 0.5 [Ste]. First, our comparison involves compilation performance: (a) Aspect weaving time in the context of both static compilation and (b) dynamic weaving, and (c) memory consumption are measured. Second, we consider the runtime impact of aspect oriented execution environments in terms of (d) the the overall performance of applications and (e) the performance of single join points in absence and presence of advice.

All measurements were performed on a Dual Xeon workstation (3 GHz per CPU) running Linux 2.4.27 with 2 GB memory. AspectJ and AspectWerkz were run on the Sun HotSpot JVM, version 1.5.0_01, and PROSE was run on version 1.4.2_08 of that VM (newer VMs do not support PROSE).

5.1 Weaving Performance and Memory Consumption

Static weaving performance is measured by compiling the Xalan-J XSLT parser [Xal] which consists of nearly 1200 classes. We have used different compilation scenarios: without any aspect (none), with an aspect advising calls of a specified method (call-one),

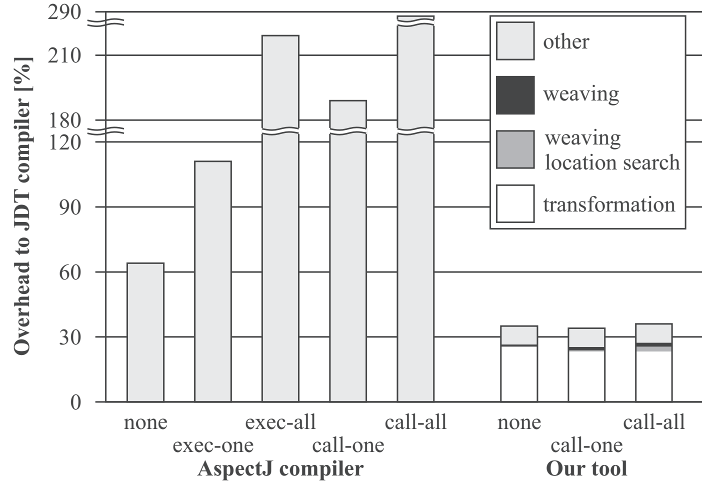

Figure 2: Compilation overheads for different compilation scenarios.

and with an aspect advising all calls[10] to public methods declared in classes in the Xalan packages (call-all). The advice simply increases a counter. In AspectJ, advice can be woven at method call sites or execution sites[11]. In the latter case the number of weaving actions is reduced, as it is in our approach. For this reason, we also mesaured `ajc`'s overhead when one specific execution site (exec-one), and executions sites of all public methods in Xalan packages (exec-all) are advised. For performance reference values, we have compiled Xalan with the Java compiler from the Eclipse JDT, version 0.452_R30x, which takes 6.4 seconds. The overheads in compilation time that `ajc` and our compiler yield when compiling Xalan for the various configurations are displayed in Fig. 2. For `ajc`, we only give the overall overhead; for our own compiler, we also provide details on how the overall compilation overhead is composed.

Like `ajc`, the overhead of our tool increases with the number of affected weavinig locations. However, for our tool the increase rate is considerably smaller. The overhead ranges from 34% in the simplest case to 36% in the most complex case. For AspectJ the overhead ranges from 64% to 289%.

Dynamic weaving performance was measured for AspectWerkz, PROSE and our prototype described in Sec. 3.6 using envelope-based weaving. We have used the RayTracer benchmark from the JavaGrande benchmark collection [Javb]. The application, consisting of 11 classes, was decorated with an aspect that advises all *calls* to public methods declared in the `raytracer` package with a simple counter-incrementing advice. For these measurements, we distinguish the time needed to *prepare* the application classes from the time for actually *deploying* an aspect at runtime. PROSE does not conduct preparation. Preparation takes 376 ms in AspectWerkz and only 66 ms in our prototype. Deployment takes

[10] Advising accesses to all fields declared in any Xalan class yields comparable results.
[11] The same is not possible for field accesses.

an additional 424 ms in AspectWerkz and 101 ms in our prototype; PROSE takes 677 ms. Again, avoiding expensive lookup operations in envelope-based weaving is beneficial.

When comparing memory consumption, our approach again benefits from the avoidance of memory-intensive data structures as AspectJ needs them during weaving. When executing scenario (call-all), `ajc` needs 121 MB of memory, while our approach only consumes 67 MB.

5.2 Runtime Overhead

To measure the *impact* of the various AOP environments on the runtime performance of applications that are not decorated with aspects, we have used the SPECjvm98 benchmarks [SPE]. They were run unmodified on the HotSpot 1.4.2 client VM and on both the client and server versions of HotSpot 1.5.0. Moreover, they were run in the AspectWerkz and PROSE environments (on HotSpot 1.5.0 and 1.4.2, respectively), and after introducing envelopes as described in Sec. 3.3 (on both HotSpot 1.5.0 VMs). Results from computing the average time spent to run the benchmarks are shown in Fig. 3.

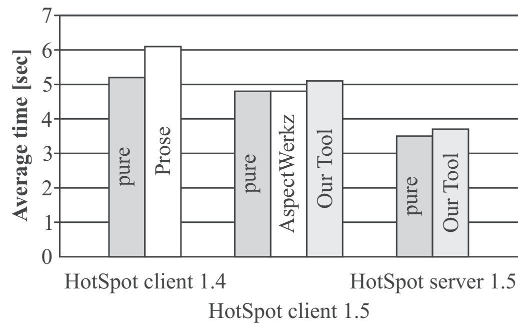

Figure 3: SPECjvm98 results.

The results indicate that envelope-based weaving and the Java Agent mentioned in Sec. 3.4 impose an overhead of 7.7 % on a running application. The overhead is however reduced to 5.2 % when an application subject to envelope-based weaving is run on the HotSpot server VM, which performs immediate JIT compilation and more aggressive inlining.

We also measured the overhead imposed simply by the presence of our Java Agent by executing the SPECjvm98 benchmarks with and without the Agent on the HotSpot 1.5.0 JVM. In average the execution was slowed down by 1.6% when run in server mode and not at all when run in client mode.

Join points' runtime performance. Finally, we were interested in the cost of attaching advice to join points. To determine this cost, we have used a suite of micro-measurements [HM] that measures the number of operations of a certain kind that a given environment can perform per second. We have applied these measurements to method calls[12], both without advice and with a (counter-incrementing) before advice attached. For these mea-

[12] We also measured field get operations. The results were similar to method calls. This is why we do not present them here.

	AspectJ	AspectWerkz	Prose	Steamloom	Our Tool
call without advice	$2.71 \cdot 10^8$	$2.69 \cdot 10^8$	$2.51 \cdot 10^8$	$1.86 \cdot 10^8$	$2.68 \cdot 10^8$
call with advide	$2.87 \cdot 10^8$	$3.04 \cdot 10^8$	$2.87 \cdot 10^3$	$9.40 \cdot 10^7$	$2.67 \cdot 10^8$

Table 1: Micro-measurement results.

surements, we have taken into account all environments listed at the beginning of this section. All systems do not suffer significantly from advising method calls, except PROSE, whose performance drop is due to the expensive context switches at debugger breakpoints (cf. Sec. 4.2).

6 Summary and Future Work

The presented weaving technique requires less time and less memory than conventional compilers for AspectJ-like languages. At the same time, the imposed runtime overhead is acceptable. As a result, this technique can be used at development time to speed up the "compile-test-debug" cycles. We have also shown that our approach can efficiently be used in runtime-weaving environments.

Our current implementation does not support the full AspectJ language. Only the pointcut designators `call`, `get`, `set`, `this`, `target`, and `within` are implemented. Also, only `before` and `after` advice is supported. Other pointcut designators, e.g., `handler`, and `around` advice can be realized similarly and will be supported in future versions of our tool. We are also working on a version with static resolution for the `within` pointcut designator.

Our current implementation does not "remembers" the actual positions of join point shadows in the source code. However, knowledge about actual join point shadows is needed by tools like Eclipse's AspectJ Development Tools (AJDT) [AJD] to present the crosscutting structure to the programmer in special views. Initial experiments, however, show that the join point shadow search can be executed efficiently during the transformation phase of our approach. During this phase, all potential join point shadows of field accesses are already visited. In the same way, it is possible to visit all potential join point shadows of method calls and match all shadows against the declared pointcuts. We prototypically extended our tool with such a join point shadow search and measured the comile time for the scenarios presented in 5.1. In the worst case this extension slows our tool down by 10.3%. What is missing is the construction of an appropriate structure for the IDE to display[13]. As a result, a combined tool that uses envelope-based weaving and conventional style join point shadow search will provide best performance.

In the current implementation of envelope-based weaving we introduce envelopes for all members and not just for those that are affected by a pointcut-and-advice. In future releases we will include this optimization which will reduce the runtime overhead. E.g., when no aspects are present in a program, the code will not be transformed at all.

[13] Such a structure can be generated by `ajc`. However, during our measurements this feature was deactivated.

As described in Sections 1 and 3.3, our weaving approach relies on the JIT compiler to optimize away the introduced indirections by inlining. In the context of runtime weaving, the same optimization might become problematic, as deploying an aspect not only entails recompilation of the affected envelopes but of all methods where envelopes are inlined. Also the methods where envelopes are inlined have to be remembered to be able to recompile them. We will investigate the possibilities of dedicated virtual machine level support and techniques like guarded inlining [DA99]. Initial experiments with the Jikes Research Virtual Machine [Jik] indicate that runtime performance can be improved this way.

The introduction of our envelopes adds methods to each class which changes the class' meta-data that is visible to the application via Java's Reflection API [Java]. It is also possible that the programmer accesses fields via the Reflection API, circumventing our accessor methods. Again, by implementing VM-level support, the behavior of the Reflection API could be changed to be compatible with the envelope approach. Similarly, the way native methods are treated can be modified to allow introduction of proxies for them.

References

[A+a] P. Avgustinov et al. Building the abc AspectJ compiler with Polyglot and Soot. `http://abc.comlab.ox.ac.uk/documents/abc-2004-4.pdf`.

[A+b] P. Avgustinov et al. Optimizing AspectJ. `http://abc.comlab.ox.ac.uk/documents/abc-2004-3.pdf`.

[A+05] P. Avgustinov et al. abc: An Extensible AspectJ Compiler. In *AOSD'05*. ACM Press, 2005.

[AJD] AspectJ Development Tools. `http://eclipse.org/ajdt/`.

[Aspa] AspectJ. `http://eclipse.org/aspectj/`.

[Aspb] AspectWerkz 2.0. `http://aspectwerkz.codehaus.org/`.

[B+99] M. Burke et al. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande Conference*, pages 81–96, 1999.

[BCE] Byte Code Engineering Library (BCEL) Manual. `http://jakarta.apache.org/bcel/manual.html`.

[BHMO] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD'04*.

[CCHW05] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley, 2005.

[DA99] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *Proc. ECOOP 1999*. Springer, 1999.

[Dmi01] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution, Proceedings (at OOPSLA 2001)*, 2001.

[Env] Envelope-Based Weaving. http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ebw/index.html.

[H+] M. Haupt et al. An Execution Layer for Aspect-Oriented Programming Languages. In *VEE'05*. ACM Press.

[HH] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *AOSD'04*. ACM Press.

[HM] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In *NODe'04*, pages 81–96.

[Hot] The Java HotSpot VM. http://java.sun.com/docs/hotspot/.

[Java] Java Reflection API. http://java.sun.com/docs/books/tutorial/reflect/.

[Javb] Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[JBo] JBoss AOP. http://www.jboss.org/products/aop.

[Jik] Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net/.

[JNI] JNI (Java Native Interface) Home Page. http://java.sun.com/j2se/1.5.0/docs/guide/jni/.

[JVM] JVMTI (Java Virtual Machine Tool Interface) Home Page. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.

[K+] G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP'97*. Springer.

[Ker] M. Kersten. AOP Tools Comparison. http://www-106.ibm.com/developerworks/java/library/j-aopwork1/.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition, 1999.

[MKC] H. Masuhara, G. Kiczales, and C.Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *CC'03*.

[PAG] A. Popovici, G. Alonso, and T. Gross. Just-in-Time Aspects: Efficient Dynamic Weaving for Java. In *AOSD'03*. ACM Press.

[PGA] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *AOSD'02*. ACM Press.

[Pol] Polyglot Parser Generator. http://www.cs.cornell.edu/Projects/polyglot/ppg.html.

[PRO] PROSE. http://prose.ethz.ch/Wiki.jsp?page=Prose.

[PSDF] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *REFLECTION'01*. Springer.

[Soo] Soot. http://www.sable.mcgill.ca/publications/papers/1999-1/sable-paper-1999-1.ps.

[SPE] SPECjvm98 Home Page. http://www.spec.org/osg/jvm98/.

[Ste] Steamloom. http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp.

[Xal] Xalan-Java version 2.6.0. http://xml.apache.org/xalan-j/.