Advice Weaving in AspectJ

Erik Hilsdale
PARC
3333 Coyote Hill Rd
Palo Alto, CA 94085
+1 650 812 4735
hilsdale@parc.com

Jim Hugunin
Want of a Nail Software
Sunnyvale, CA
+1 650 812 4735
jim-aj@hugunin.net

ABSTRACT

This paper describes the implementation of advice weaving in AspectJ. The AspectJ language picks out dynamic join points in a program's execution with pointcuts and uses advice to change the behavior at those join points. The core task of AspectJ's advice weaver is to statically transform a program so that at runtime it will behave according to the AspectJ language semantics. This paper describes the 1.1 implementation which is based on transforming bytecode. We describe how AspectJ's join points are mapped to regions of bytecode, how these regions are efficiently matched by AspectJ's pointcuts, and how pieces of advice are efficiently implemented at these regions. We also discuss both run-time and compile-time performance of this implementation.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Implementation and compilers

General Terms

Performance, Design, Languages

Keywords

Aspect-orientation, bytecode, AspectJ, compilers, weaving

1. INTRODUCTION

There have been three books [9],[5],[11] and numerous articles [7],[8] written about the AspectJ language [1], but this is the first discussion of the implementation concerns for AspectJ. Every AOP language implementation must ensure that aspect and non-aspect code run together in a properly coordinated fashion. A central part of this coordination, advice weaving, ensures that advice runs at the appropriate join points as specified by the program. This paper describes the implementation of advice weaving in AspectJ.

AOSD '04, March, 2004, Lancaster, UK.

We begin with an overview of the AspectJ 1.1 compiler. Then move on to discuss join point shadows, the process of matching advice to join point shadows, and the process of implementing advice at matched shadows. We finish with performance results and benchmarks, and a discussion of related work.

This paper does not cover the entire AspectJ compiler. AspectJ contains other crosscutting features than advice, such as inter-type declarations, **`declare parents`**, **`declare soft`**, **`privileged`** aspects and non-singleton aspect instances (**`per*`** aspects).

We do cover our implementation of **`declare error`** and **`declare warning`**, which are implemented through the advice framework.

2. THE COMPILATION PROCESS

The AspectJ compiler accepts both AspectJ bytecode and source code and produces pure Java bytecode as a result. Internally it has two stages. The front-end compiles both AspectJ and pure Java source code into pure Java bytecode annotated with additional attributes representing any non-java forms such as advice and pointcut declarations. The back-end of the AspectJ compiler–the part that the majority of this paper covers–implements the transformations encoded in these attributes to produce woven class files. The back-end can be run stand-alone to weave pre-compiled aspects into pre-compiled .jar files. In addition, the back-end exposes a weaving API which can be used to implement ClassLoaders that will weave advice into classes dynamically as they are loaded by the virtual machine. This API has been used to implement dynamic weaving of classes within the eclipse IDE [13].

2.1 Source File Compilation

The front-end of the AspectJ compiler is implemented as an extension of the Java compiler from eclipse.org [6]. The source-file portion of the AspectJ compiler is made complicated by inter-type declarations, **`declare parents`**, **`declare soft`**, and **`privileged`** aspects. All of these constructs require considerable changes to the underlying compiler to modify Java's name-binding and static checking behavior.

However, the compilation of advice is relatively simple. Each advice declaration is compiled into a standard Java method. The parameters of this new method are the parameters of the advice, possibly extended with thisJoinPoint reflective information as described in section 2.1.1. The body of the method is the same as the body of the advice with special handling for **`proceed`** in around advice described in section 2.1.2.

For example, the following advice declaration

```
before(String s): execution(void go(*)) && args(s)
{
    System.out.println(s);
}
```

is compiled into a method encapsulating the body of the advice

```
public void ajc$before$A$a9();
0: getstatic [java/lang/System.out]
3: aload_1
4: invokevirtual [java/io/PrintStream.println]
7: return
```

The method is annotated with an additional attribute to indicate that this corresponds to an advice declaration, to store the pointcut referred to by the advice, and to store additional information relevant to thisJoinPoint and **proceed**. For example, our implementation of around advice needs to know if the advice's body contains a call to **proceed** inside a nested type; the front end transmits that information to the back end through this attribute. This attribute is encoded as a standard Java bytecode attribute [JVM v2 4.7] to be compatible with all JVMs.

All of the parameters to the advice are statically typed and the weaving process guarantees that the advice will only be called with appropriate values. This static type checking in the face of dynamic matching is an important property of AspectJ's compilation model.

2.1.1 thisJoinPoint and variants

Within the body of an advice declaration, three special variables are exposed that can be used to reflectively discover both static and dynamic information about the current join point. The most common use of these variables is in tracing and logging applications. At compile-time, these variables are implemented by extending the signature of the advice method with three additional parameters:

```
(_, JoinPoint thisJoinPoint,
 JoinPoint.StaticPart thisJoinPointStaticPart,
 JoinPoint.StaticPart
             thisEncolsingJoinPointStaticPart)
```

This reflective information can be very expensive to compute. The most expensive part is the creation of an object array to hold the args, which may need to be converted from primitive values. This overhead, however, need only be present when the advice requires the information. So an important performance optimization the AspectJ compiler performs is to remove from the signature any special variables that are not referred to within the body of the advice.

A separate optimization will determine if all uses of thisJoinPoint can be replaced with thisJoinPointStaticPart. If this is true, then references to thisJoinPoint are replaced with references to thisJoinPointStaticPart to avoid the creation of a JoinPoint object at every matching dynamic point in the program's execution.

2.1.2 proceed

Around advice in AspectJ uses the special form **proceed** to continue with the normal flow of execution at the corresponding join point. In the front-end, this special form is implemented by generating a method which takes in all of the original arguments to the around method plus an additional AroundClosure object that encapsulates the normal flow of execution which has been interrupted by the advice. The body of the **proceed** method will call a method on the AroundClosure to continue with the execution, passing in any parts of the state that have been modified by the around advice.

2.2 Bytecode transformation

The back-end of the AspectJ compiler instruments the code of the system by inserting calls to the precompiled advice methods. It does this by considering that certain principled places in bytecode represent possible join points; these are the "static shadow" of those join points. For each such static shadow, it checks each piece of advice in the system and determines if the advice's pointcut could match that static shadow. If it could match, it inserts a call to the advice's implementation method guarded by any dynamic testing needed to ensure the match.

For example, consider the simple before advice in 2.1. The pointcut on this advice specifies that it should be run before the execution of every void method named **go** when called with a single String argument. The weaver must instrument the bytecode to insert calls to the before advice whenever this condition is met.

For example, consider the following empty method:

```
void go(java/lang/Object):
0: return
```

The weaver can statically determine that an execution of this method may match the pointcut and cause the advice to execute. However, a dynamic test must be inserted to check whether or not the type of the argument to the method is a String and only invoke the before advice when this condition is met. This will produce the following woven code:

```
void go(java/lang/Object):
0:  aload_1     # defensive copy of first argument
1:  astore_2    # into temporary frame location
2:  aload_2               # check whether argument
3:  instanceof [String]   # is actually a String
6:  ifeq 19           # if not, skip advice
9:  invokestatic  [A.aspectOf]        # get aspect
12: aload_2      # load argument again
13: checkcast [String]  # guaranteed to succeed
16: invokevirtual [A.ajc$before$A$a3] # run advice
19: return
```

In order to achieve good performance in woven code, the weaver needs to eliminate these kinds of dynamic checks whenever possible. For example, if the static type of **go**'s parameter is Number, the weaver can statically determine that the pointcut can never match this join point and no code will be woven. On the other hand, if the static type of **go**'s parameter is String then the dynamic tests can be eliminated resulting in a direct call to the advice:

```
void go(java/lang/String):
0: invokestatic  [A.aspectOf]
3: invokevirtual [A.ajc$before$A$a3]
6: return
```

AspectJ advice is always run in the context of an aspect instance—thus the advice method is non-static. By default aspect instances are singletons accessed through the **aspectOf** static method. AspectJ does allow non-singleton aspect instances through **per*** clauses, but they are beyond the scope of this paper.

We will describe the matching and weaving processes in more detail in sections 4 and 5 respectively.

Table 1. Join point shadow kinds

kind	signature	this	target	args	Bytecode shadow
Method-execution	method	ALOAD_0 or none	Same as this	Local vars	Entire code segment of method
Method-call	method	ALOAD_0 or none	From stack	From stack	Invokeinterface, invokespecial *(only for privates)*, invokestatic, invokevirtual
Constructor-execution	constructor	ALOAD_0	Same as this	Local vars	Code segment of <init> after call to super
Constructor-call	Constructor	ALOAD_0 or none	None	From stack	Invokespecial *(plus some extra pieces)*
Field-get	Field	ALOAD_0 or none	From stack	none	Getfield or getstatic
Field-set	Field	ALOAD_0 or none	From stack	From stack	Putfield or putstatic
Advice-execution	None	ALOAD_0	Same as this	Local vars	Code segment of corresponding method
Initialization	Corresponding constructor	ALOAD_0	Same as this	Complex	Requires in-lining of all constructors in a given class into one
Static-initialization	Typename	None	None	None	Code segment of <clinit>
Pre-initialization	Corresponding constructor	None	None	Local vars	Code segment of <init> before call to super, this may require in-lining
Exception-handler	Typename of exception	ALOAD_0 or none	None	From stack	Start is found from exception handler table. *(only before advice allowed because end is poorly defined in bytecode)*
Exception-throws (not in AspectJ-1.1)	Typename of exception	ALOAD_0 or none	None	From stack	athrow
Synchronized-block (not in AspectJ-1.1)	Typename of lock object	ALOAD_0 or none	None	From stack	Code between monitorenter/monitorexit pair

3. Join point shadows

In the AspectJ language model, a join point is a point in the dynamic call graph of a running program where the behavior of the program can be modified by advice. Every dynamic join point has a corresponding static shadow in the source code or bytecode of the program. The AspectJ compiler inserts code at these static shadows in order to modify the dynamic behavior of the program. The other possible implementation would be to modify the JVM or to use runtime hooks as provided by the debugging APIs to more directly match the dynamic join point model in the AspectJ language.

3.1 What defines a join point shadow?

There are 11 kinds of join point shadows in AspectJ-1.1, and these are all shown in Table 1. That table also includes two join points that have been proposed for future versions of the language to show the extensibility of the model.

Every join point shadow is defined by a kind, a signature, and a region of bytecode. Each shadow also has a source location that is given by the SourceFile attribute of the enclosing class file [JVM v2 4.7.7], and whose line number is determined from the LineNumberTable attribute [JVM v2 4.7.8].

Almost all join point shadows can be clearly defined in terms of a bounded region of bytecode. There are a few exceptions. Initialization shadows require all constructors within a method to be inlined in order for their bytecode segment to be correct. This is done lazily only if needed. In addition, exception-handler shadows do not have a clearly defined end-point.

Each join point also exposes up to three pieces of state.

- this – the currently executing object. In a static context this is always none, and in an instance context this can always be found with aload_0.
- target – the target object of the join point
- args – the arguments to the join point

This state can be used for matching and exposed by the pointcut designator language in AspectJ. It can also be accessed through a reflective object which exposes only the static part of the context (thisJoinPointStaticPart) or all of the state including this, target and args (thisJoinPoint).

3.2 The shadows of "hello world"

Let's look at the bytecode for a simple hello world program:

```
0: getstatic [java/lang/System.out]
3: ldc       [String hello world]
5: invokevirtual [java/io/PrintStream.println]
8: return
```

There are three shadows in this short program. The previous section discussed the method-execution shadow which is wrapped around all of the code above from #0-#8. Next is the field-get shadow which is represented by the single bytecode at #0. The third shadow is for the method-call at #5.

Weaving advice into this method-call shadow is straight-forward and very similar to weaving into a method-execution shadow as described before. In this case, the inserted call must go before the `invokevirtual` instruction rather than at the beginning of the method, e.g.

```
0: getstatic [java/lang/System.out]
3: ldc       [String hello world]
  5: invokestatic  [A.aspectOf]
  8: invokevirtual [A.ajc$before$A$15a]
11: invokevirtual [java/io/PrintStream.println]
14: return
```

Here the call to the advice has been inserted just before the **invokevirtual** instruction corresponding to the shadow.

3.2.1 Exposing state for method-call

Exposing state at a method call is slightly more difficult because the target and args are sitting on the stack rather than in local variables. If we need to expose this state, then we will need to pull this information off of the stack into temporary variables, use these variables to provide the state to the advice, and then push these variables back onto the stack to make the original call, i.e.

```
0: getstatic [java/lang/System.out]
3: ldc       [String hello world]
  5: astore_1
  6: astore_2
  7: invokestatic  [A.aspectOf]
  10: aload_2
  11: invokevirtual [A.ajc$before$A$15a]
  14: aload_2
  15: aload_1
16: invokevirtual [java/io/PrintStream.println]
19: return
```

- 5-6 – store the contents of the stack in local variables
- 10 – load the target object for the call to the advice
- 14-15 – push the local variables back onto the stack for the call

Optimization note: The size of this code could be reduced if we recognized that the argument to the method is a constant value. This standard analysis may be worth doing in the future. However, this optimization would need to be very conservative. For example, the **getstatic** operation can not be moved as it has potential side-effects if the class the field is on has not yet been initialized. Any changes from this simple model must not be observable without inspecting the bytecodes themselves.

4. Matching

Advice and other advice-like entities are represented by shadow munger objects. A shadow munger performs transformations on join point shadows matched by its contained pointcut designator (PCD). There are shadow mungers for all 5 kinds of advice, **declare error**, **declare warning**, **declare soft**, control flow entry and exit, and **per*** aspect creation.

During the weaving process, the PCD for each shadow munger is matched against each join point shadow in the bytecode being processed. Because the AspectJ language defines a join point as a dynamic point in the call graph of the running program, the matching process might not be completely statically resolvable. When the PCDs depend on the dynamic state at the join point this mismatch is resolved by adding a dynamic test that captures the dynamic part of the matching. We call this dynamic test the residue of the match.

4.1 Residues

4.1.1 If residue

The **if** PCD specifies an arbitrary expression to evaluate at each join point. Static matching against this PCD is always true and results in nothing but a dynamic residue. The residue is implemented by compiling the if expression into a static test method in the type declaring the PCD. If any join point state is required for the test, that state is passed into the new static method as arguments. The PCD then resolves to a dynamic test which will call the corresponding method. For example, the advice

```
before(): execution(void main(*))
          && if(Tracing.level == 1) {
    System.out.println("got here");
}
```

when applied to our simple hello world program will result in

```
0:  invokestatic [A.ajc$if_0]  # dynamic test
3:  ifeq 12
  6:  invokestatic [A.aspectOf]
  9:  invokevirtual [Method A.ajc$before$A$a6]
12: getstatic [java/lang/System.out]
15: ldc ["hello world"]
17: invokevirtual java/io/PrintStream.println]
20: return
```

An interesting area for future work would be to consider partial evaluation optimizations for more precise static matching.

4.1.2 Instanceof residues

The three PCD's **this**, **target** and **args** all define matching based on the dynamic type of the state exposed at a join point. All three of these potentially add a dynamic **instanceof** test to join point shadows that they matched. As we explained in section 2.2, this residue is only generated when we can't statically determine that a match will always or never succeed.

4.1.3 Cflow residue

The **cflow** PCD in AspectJ allows matching join points that are within the dynamic control-flow of other join points. In AspectJ-1.1 this matching is implemented entirely as a dynamic test on the join point. No static analysis is performed to try to determine whether a **cflow** match is either always or never possible. It is unclear that such static analysis could be useful in anything except the simplest examples without requiring whole-program analysis. And even with whole-program analysis, there are cases in the presence of reflective calls where static analysis of **cflow** is impossible.

The current implementation of **cflow** uses a thread-local stack to keep track of the entries and exits of join points that match the predicate join point. This stack is updated by a shadow munger similarly to how advice is woven, see 5.2.5. Matching of a **cflow** PCD is then a simple matter of testing the appropriate thread-local stack. If state is exposed by the **cflow** PCD, that state is stored in the same thread-local stack described above in CFlowPlusState objects.

4.2 Fastmatch

Matching every join point shadow in every class file can be a time consuming process. Section 6 shows that just this matching process can more than double the time to compile a large system. This performance issue can be even more significant for load-time weaving where this matching overhead can be visible to the user.

AspectJ-1.1 uses a fastmatch pass to improve matching time. In this pass, every shadow munger is matched to the constant pool information in each class file. This information can be computed very cheaply so this is an extremely fast process. Currently, the only PCD for which fastmatch is implemented is `within`. This is the easiest possible case that can be determined solely from the fully-qualified name of the class being matched against. This is an area where we expect to see significant improvements to dramatically reduce weaving times in future versions of AspectJ.

Another unimplemented optimization would be to expand the notion of fastmatch to determine which kinds of join points could be matched by the valid shadow mungers. For example, if the current shadow mungers can only apply to method-execution join points, then performance could be significantly improved by never considering all of the 10 other kinds of join point shadows that will be present in the class file.

4.3 Synthetic methods and matching

Compilers for the Java language generate methods and fields that are not in the original source code. This is done primarily for `assert`, the `.class` expression, and inner class implementation. These synthetic constructs are not considered join points in the AspectJ language model. Therefore the implementation needs to use the SYNTHETIC attribute in the Java bytecode [JVMS v2 4.7.6] to recognize and exclude these potential join point shadows.

There are many additional synthetic methods added by the AspectJ compiler that are not described in this paper. These include the implementation of inter-type declarations and `per*` aspects. These constructs are labeled with the AJ_SYNTHETIC attribute that may also specify an effective signature that should be used to represent them for the purposes of PCD matching.

5. Weaving

Once the weaver has matched various shadow mungers to each join point shadow, the mungers themselves are implemented in two stages. First, any context that is needed by any of the shadow mungers is exposed at the join point. Next, each shadow munger is applied to the shadow, changing the bytecode to implement the desired behavior. There are many different kinds of shadow mungers, each with a different kind of expansion.

5.1 Context exposure

First, all the shadow mungers are queried to determine what state they need and the bytecode is modified to expose that state into local variables. The state needs is that exposed by the `this`, `target` and `args` PCDs. For join point shadows where this state is on the frame (see Table 1. Joinpoint shadow kinds) this simply involves making a copy of frame contents.

When the arguments are on the stack instead, they must be popped into local variables and then pushed back onto the stack to make the original call. An example of this transformation for a method-call join point is shown in section 3.2.1. This exposure step results in a join point shadow equivalent to the original program except that all state exposed by `this`, `target`, and `args` PCDs is in frame locations.

If any of the shadow mungers refer to thisJoinPoint, then this exposure step will also include the creation of a new JoinPoint instance for this join point. This object will be created with all of the state from this, target and args.

Note that this context exposure is done once and only once for each matched join point shadow, regardless of the number of shadow mungers that have matched it.

5.2 Shadow Munger Implementation

Once a shadow is transformed to expose context, each shadow munger that matched the shadow is implemented in turn. Each shadow munger transforms the shadow by adding code inside the boundary of the shadow. This means that after one shadow munger is implemented, the next shadow munger to be implemented on the same shadow will encapsulate the shadow including the advice; thus, pieces of advice must be woven in inverse precedence order.

5.2.1 Before Advice

The simplest application is before advice. It is a property of a join point shadow that there is only one entry point to the shadow: shadows in switch statements do not, for example, split across multiple case lines. This means that before advice is comparatively easy: We simply insert the call to the advice method at the beginning of the shadow. We've already shown many examples of weaving before advice in earlier sections of this paper.

5.2.2 After Returning Advice

After returning advice is more difficult, because there are potentially many exit points from a shadow. There are two cases. The first is that there is exactly one exit point from the shadow, the fallthrough to the next instruction. This is the case for shadows of the call join point, for example. The other case is that there are potentially many exits, but all are return bytecodes (`return` or one of its typed variants such as `ireturn`). This is the case for shadows of the execution join point, among others.

After returning advice on the first case simply involves inserting code to possibly expose the return value, if needed, and then calling the advice.

In the second, more complicated, case we first process the join point shadow, converting the return bytecodes into `goto`s jumping to an inserted return bytecode at the end of the shadow. We then insert the advice call at the single return bytecode. This avoids duplicating the advice invocation code. Figure 1 shows the effect of weaving a piece of after returning advice on an implementation of factorial with two `ireturn` bytecodes.

5.2.3 After Throwing Advice

While a join point shadow may have many exit points, for after throwing advice the only ones we care about are abrupt exits with an exception. These are simply captured by adding a new entry to the enclosing method's exception handler table for the bytecode region corresponding to the shadow. The code for the handler contains the call to the advice and is inserted at the end of the shadow. A `goto` is inserted to branch around the handler when necessary.

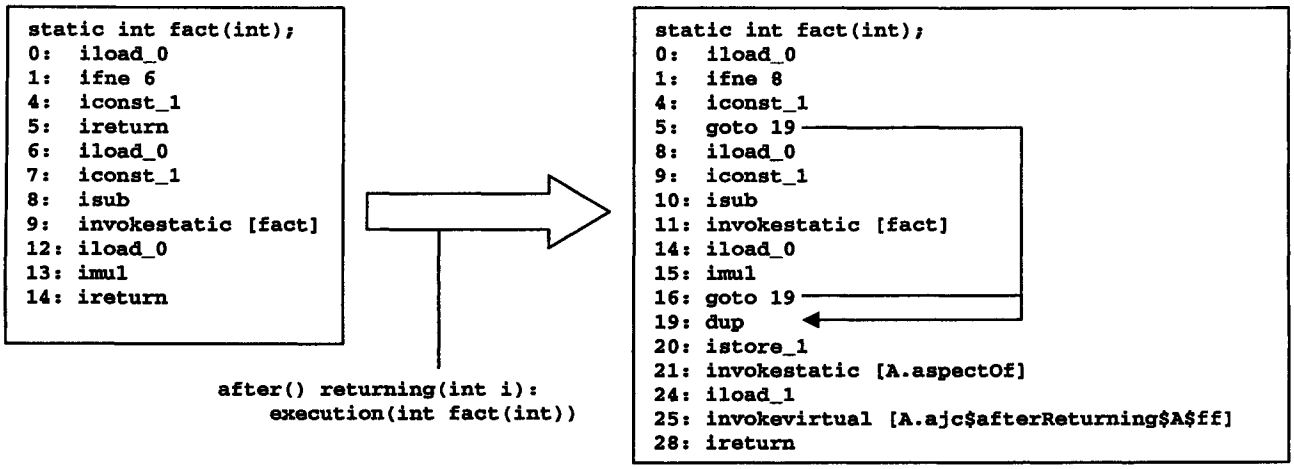

Figure 1. After Returning Advice

5.2.4 After Finally Advice

After finally advice represents advice that should run both after returning and after throwing. In previous versions of AspectJ, this was implemented by a finally block that was compiled using `jsr` and `ret` bytecodes to avoid code duplication. In AspectJ 1.1, having already abstracted the advice instructions into methods, we are able to implement after finally advice simply as the composition of after returning and after throwing advice, duplicating the call to the advice but not the advice itself.

5.2.5 Control Flow entry and exit

The implementation of the `cflow` pointcut requires a single shadow mungers for the entry and exit from a particular join point. Control flow entry is implemented as if it were before advice and control flow exit as if it were after finally advice. Instead of calling an advice method, however, the action taken is to manipulate a control-flow stack. Note that this must be implemented as a single shadow munger that performs any dynamic tests all at the same time before entering the join point and stores the result in a local boolean variable. If the dynamic test was performed on both entry and exit it might have different results leaving the control-flow stack in an inconsistent state.

5.2.6 Around Advice

The most complicated advice implementation is for around advice. This is because around advice must completely encapsulate its join point shadow into its `proceed` call.

Like other kinds of advice, an around advice declaration is compiled into a method, taking as arguments any advice parameters. In addition, though, it takes one additional argument, an AroundClosure object. Any calls to `proceed` in the body of the advice are represented by calls to a `run` method on that AroundClosure. Recall also that the front end sets an attribute as to whether there was a call to `proceed` from within a nested type.

When around advice is woven at a particular shadow the bytecode for the shadow is first extracted into its own method, accepting as arguments any free variables from the enclosing method. How that method is called depends on whether the advice had a call to proceed from a nested type.

If there was a proceed call in a nested type the weaver must assume that the call to `proceed` is closed over. Therefore we need to create a closure object for the proceed call. We create a new subclass of AroundClosure whose run method dispatches to the new shadow method. In place of the shadow is left code that instantiates an instance of the new subclass and passes that instance to the around advice (in addition to any state the around advice requires).

If, however, we do not need to create a closure object, we instead inline the around advice. This involves copying the code for the advice method replacing the call to the AroundClosure's `run` method with a call to the extracted shadow method. This avoids not only the runtime cost of storing closure state, but also the cost of generating a new class.

5.2.7 Declare warning and error

These are matched to join points in the same way as advice. However, instead of modifying the bytecode for the join point, they generate a message to the user indicating either a warning or an error. There is a restriction in the language that these constructs can't use any PCDs that could produce a dynamic residue as part of their matching process.

5.3 Why we don't inline advice code

The primary performance overhead of AspectJ code is caused by the aspect-instance lookup and method call. This overhead could be eliminated if the weaver would inline the advice code directly into the join point. Previous versions of AspectJ used this implementation strategy.

Inlining can be done much more effectively by a JIT than by a tool that has to follow Java's access rules. Code within an aspect must follow Java's standard lexical accessibility rules. If code within advice access private members on the aspect, or package-visible members in the aspects package, these members would not normally be visible if the code is moved to another class in a different package. In order to address this problem we would need to either increase the visibility of the accessed members, which would let anyone see them and could even break our inheritance hierarchy. We could add synthetic accessor methods with mangled names to expose these fields; this would be very

similar to the approach taken for implementing Java's inner classes, except it has the additional concern that it must often expose members to other packages. A more sophisticated authentication scheme could probably be devised with even more performance and implementation overhead.

We do use this inlining strategy for the default (non-closure) implementation of around advice; as well as for privileged aspects which are allowed to access other type's members without respecting Java's accessibility rules. In many cases the performance overhead of these accessor methods would destroy any gains achieved by inlining the advice code originally.

Given the ever-improving quality of JITs for Java, inlining optimizations must be carefully considered. Without careful benchmarking there is as much potential for reducing the performance of the resulting system as for improving it.

6. Compile time performance

Because an AspectJ compiler (ajc) needs to do more work than a pure Java compiler, we expect that it will take longer to compile a system. This section measures the overhead in compile time introduced by using ajc vs. the standard javac compiler from Sun.

We chose to measure performance of one large system rather than several small ones. Given the extreme and dynamic optimizations performed by modern JITs, we believe that performance measurements on a real application will most accurately reflect the performance seen by AspectJ developers in practice. We chose the logging aspect as it is the most invasive of the widely used AspectJ aspects.

We measure the time required to compile the xalan xslt processor from apache.org [16] under both ajc and several versions of the standard java compiler from Sun. This system has 826 source files with 144,631 non-comment/non-blank lines of code. All tests in this section and the next were run were run on a 1.4GHz Pentium-M processor with 768MB of RAM under SUN's j2sdk-1.4.2.

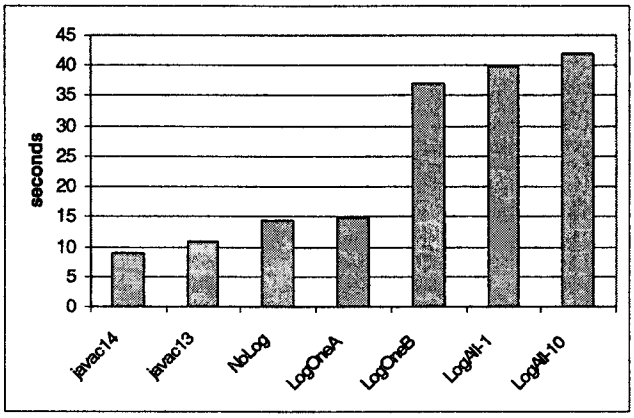

Figure 2. Compile time for xalan with different aspects

The first three results show that ajc is about 62% slower than the 1.4 javac compiler and 34% slower than the 1.3 javac compiler when compiling pure java code. This extra performance overhead is caused by the need to analyze the generated classes to see if any advice might need to be woven into them. It's expected that this overhead can be reduced in future releases by straight-forward engineering work.

For the next tests we add different aspects to the system. The first aspect contains a single piece of advice that adds logging to exactly one method in xalan (LogOneA). This advice uses the within PCD to make explicit the single class that it can apply to. Because of the efficient fastmatch code described in section 4.2, adding this aspect introduces barely any overhead over the compile with no aspects.

In the next test, we modify the advice so that it doesn't use the `within` PCD (LogOneB). This advice still applies only to a single method in the xalan code, but because this isn't recognized by the fastmatch code the advice must be matched against every join point shadow in xalan. This minor change results in a major 256% increase in compile time.

The final two tests modify the advice to add logging to all methods in xalan. They do this with either one (LogAll-1) or ten different (LogAll-10) pieces of advice for each method execution join point shadow. These changes to actually weave advice add relatively small 8% and 13% additional overhead compared to the previous test that did minimal weaving but required matching for all join point shadows in the code. This shows that the most important performance bottleneck in the AspectJ 1.1 weaver comes from collecting all of the join point shadows for matching against any viable shadow mungers.

There are three main ways to improve compile-time. Basic engineering work should be able to reduce all of the overheads in a similar way to the 21% performance gain Sun achieved from javac-1.3 to javac-1.4. Clearly, improving the fastmatch algorithm to handle more cases could have a dramatic impact on the compile-time when aspects don't affect a large fraction of the classes in a system. The final approach to improving performance is to support incremental recompilation so that the whole system isn't rebuilt every time. AspectJ 1.1 has support for an incremental compilation mode, but further investigation of that subject is beyond the scope of this paper.

7. Performance of woven code

The implementation of advice weaving introduces very little performance overhead when compared to the same functionality coded by hand. The current weaving implementation adds a static field lookup and a call to a final method compared to a hand-coded implementation that would inline the advice code. This section will explore the addition of an aggressive logging policy to a realistic system in order to assess the performance overhead of advice in AspectJ 1.1.

7.1 Basic benchmark

Logging as an aspect is frequently used as an example of the advantages of AOP. A standard implementation of logging requires the modification of virtually every method in a system by hand. AOP implementations instead capture this policy in one place. On the other hand, logging is also the kind of application where performance overhead is most important. Because logging policies affect a huge number of methods in a system, even a small performance impact will be noticeable in overall system performance.

For this example we will use a simple but very aggressive logging policy to log all method entries in the xalan code base but not in the libraries that it uses. We will use the standard logging API available in j2se-1.4 to implement the actual logging.

We measure the performance impact of these changes with the XSLTMark benchmark [17] that is often used to compare the performance of different XSLT implementations. Our performance numbers measure the execution time for the entire XSLTMark benchmark suite with the exception of 5 test cases that were found to fail (dbonerow, html, xslbench1, xslbench2, xslbench3).

7.1.1 Hand-coded implementation

As a base-line, we used a standard bytecode manipulation toolkit to produce a version of the code corresponding to a hand-coded implementation of this kind of logging policy. This implementation is equivalent to making the following changes to the code by hand.

The following static field is added to each of the 826 classes:

```
static Logger log = Logger.getLogger("xalan");
```

and a call is added to the beginning of each of the 7711 methods:

```
log.entering("<ClassName>", "<MethodName>");
```

The logging API promises that the **Logger.entering** method will run extremely fast in the case that logging is disabled so no guard method to check whether or not logging is enabled is needed and adding such a check does not have a noticeable impact on performance.

7.1.2 A naïve AspectJ implementation

We also wrote the simplest possible aspect that could capture this same logging policy. It uses before advice to make the same `log.entering` call at the entrance to every method in the xalan code base.

```
public aspect Trace {
  private static Logger log =
    Logger.getLogger("xalan");

  pointcut traced():execution(* *(..));

  before(): traced() {
    Signature s =
     thisJoinPointStaticPart.getSignature();

    log.entering(
      s.getDeclaringType().getName(),
      s.getname());
  }
}
```

7.2 Initial Performance Overhead Results

With logging enabled, there is a little more than a 600X performance slow-down to the application. The AspectJ implementation is about 3% slower than the hand-coded implementation. This difference is barely noticeable. However, this is not the situation in which people are typically concerned about the performance overhead of a logging implementation. The important case is to consider the overhead when logging is disabled.

With logging disabled we find a huge performance overhead for the naïve AspectJ implementation. This 2900% overhead is equivalent to 2221 nsec per method execution.

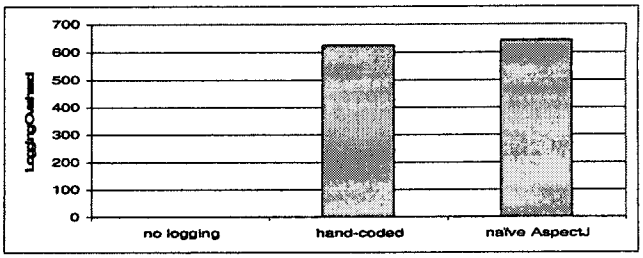

Figure 3. Overhead with logging enabled

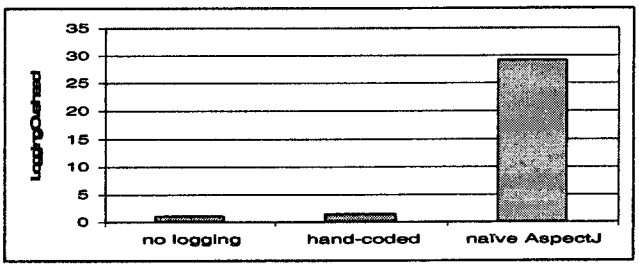

Figure 4. Overhead with logging disabled

7.2.1 The performance costs of Class.getName()

The main performance overhead in a naïve AspectJ implementation of tracing is the cost of calling `Class.getName()`. This was a surprising result and indicates a deficiency in AspectJ's reflection API. In AspectJ-1.2, we will add a method `Signature.getDeclaringClassName()` to work around this deficiency in the underlying Java reflection implementation.

Because `Class.getName()` is a native method, its implementation is unavailable to most Java developers. To better understand the performance issues, we implemented a very simple test which called the method a large number of times for classes whose names had varying lengths. We discovered the following simple behavior, each call has a fixed overhead of 325ns with an additional overhead of 43ns per character in the classes name. This suggests that a new String object is being constructed for each call to this method. We can also conclude from this that the average length of class name in the xalan code is about 45 characters.

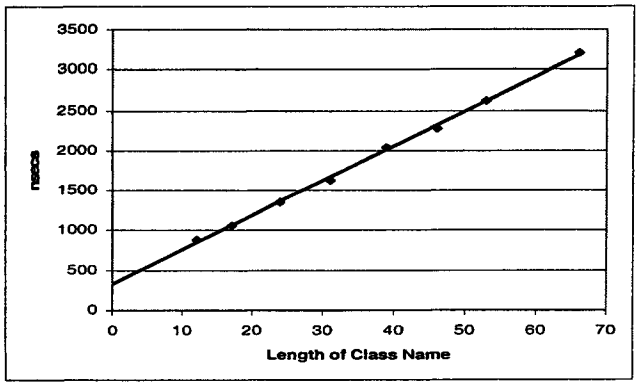

Figure 5. Time for one call to Class.getName()

7.3 More efficient AspectJ implementations

It's clearly important to avoid calls to `Class.getName()` for an efficient AspectJ implementation of tracing. We can do this by calling the `log.loggable` method before calling `Class.getName()` to prepare the arguments for the call to `log.entering`. This one-line change will reduce the overhead of the AspectJ implementation from 2500% to a more reasonable 82% compared to the hand-coded implementation.

```
before(): traced() {
    if (!log.isLoggable(Level.FINER)) return;
    ...
}
```

Further performance improvements to the AspectJ code can be found by putting the `log.loggable` test in an `if` PCD. This will put the check in a single static method further improving performance.

```
pointcut traced(): execution(* *(..)) &&
  if (log.isLoggable(Level.FINER));
```

This is the fastest AspectJ implementation that doesn't change or restrict the tracing policy in any way. The performance overhead of this implementation is just 22% greater than the overhead caused by the hand-coded implementation and is unlikely to be noticeable in most applications.

If the remaining performance overhead is still an issue, the modular implementation made possible by AspectJ makes it easy to consider small changes to the tracing policy that can improve performance considerably. One option is to add a static enabled field to the aspect that must be set in addition to the normal logger API calls to enable logging.

```
static boolean enabled;
pointcut traced(): execution(* *(..)) &&
  if (enabled) && if(log.isLoggable(Level.FINER));
```

If this field is used, then the AspectJ implementation will perform better than the standard hand-coded logging implementation by a 76% margin. Obviously, this same test could be added to the hand-coded logger, but that would require modifying and maintaining this modification at 7700 different places in the code.

The most extreme optimization that AspectJ makes possible is to completely remove the logging code from the system by not including the aspect in a high-performance build. This will result in absolutely no performance overhead, with the consequence that logging can not be turned on dynamically at runtime.

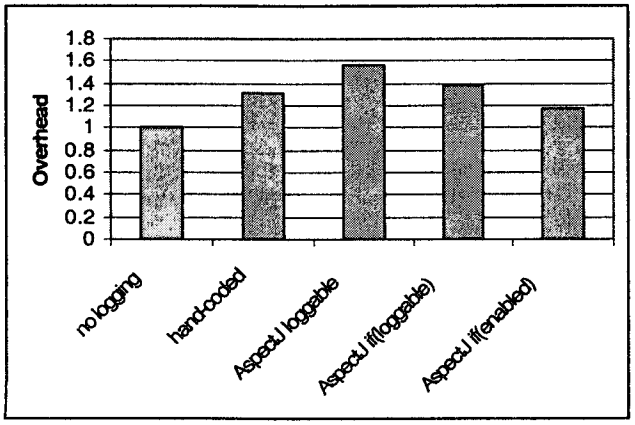

Figure 6. Overhead of efficient AspectJ implementations

7.4 Performance Conclusions

The best AspectJ implementation of logging adds a 22% overhead relative to the hand-coded logging implementation. This is an upper bound on the performance overhead for well-written advice because there is almost no work taking place within the body of the advice itself. More complicated aspects will have considerably less overhead as more time is spent in the advice and less in the dispatch process.

We also found that we could easily modify the AspectJ code to experiment with different logging designs. This led to a design which incorporates an additional static boolean field. Implementing this change in AspectJ required changing 2 lines of code vs. 7700 that would be required to change in the hand-coded implementation. This slightly altered logging policy has 76% less overhead than the hand-coded implementation. This ability to quickly experiment with different designs means that systems built with AspectJ can often have better performance in practice than their less flexible hand-coded counterparts.

The small performance overhead found in the current AspectJ implementation could be eliminated by performing aggressive inlining during weaving. However, there are many reasons why we have decided not to implement this inlining in the current release of AspectJ. These are discussed in section 5.3. However, these sorts of optimizations will probably appear either in JIT improvements or in future versions of AspectJ and other AOP systems. As AOP matures, the performance overhead for well-written aspect code should fall close to zero.

Nevertheless, our performance measurements revealed that AspectJ gives programmers the ability to write extremely inefficient code quickly and easily. The naïve logging implementation showed a 2900% performance overhead. This is always a danger with new and powerful tools. As is the case with any programming language, addressing the potential performance pitfalls of poorly written code will depend more on education than on technical improvements to the tools themselves.

8. Related Work

There are many different ways to implement advice for Java. The two primary approaches are either to generate a transformed Java program with the advice semantics encoded in it or to modify the virtual machine to provide additional hooks at run-time.

The Just-In-Time aspects project [15] is one of the few that modifies a JVM to directly implement advice semantics. These approaches hold promise for very dynamic weaving support at some stage in the future; however, they currently require substantial performance overheads to be used. The greatest practical performance overhead of this approach is that the implementations only run on research JVMs which are noticeably slower than the latest production machines.

There are two main approaches to transforming a Java program to implement advice. One approach is to insert generic hooks at all possible join point shadows at transformation-time. This approach allows for specific advice to be dynamically added or subtracted from these points at run-time. This is the approach taken in JBoss [3] and Handi-Wrap [2]. The primary drawback of this implementation strategy is that it adds some measurable overhead to every join point shadow that it exposes. JBoss reduces this impact by using a coarse-grained join point model that only captures method and constructor executions. However,

this performance impact would be very substantial if used for all of AspectJ's fine-grained join points.

Hyper/J [14] was the first AOSD system to be implemented by transforming existing Java bytecodes.

AspectJ's implementation transforms a Java program in the presence of a particular set of advice. The weaving process then only inserts code at those join point shadows that could be matched by some advice. It is still possible in this system to enable and disable aspects efficiently at run-time. The "if(enabled)" version of the logging aspect in section 7.3 is a good example of this.

AspectJ's implementations have used every form of transformation imaginable for a Java program. The earliest versions operated as preprocessors using javac as a back-end. The 1.0 version of AspectJ could operate as both a preprocessor and a full source-code compiler. The 1.1 version described in this paper operates as a bytecode transformer.

The AspectJ story has always been one of being a language rather than a meta-language or transformation framework. The current *implementation* of AspectJ, however, shares many properties with such frameworks and meta-languages. One such framework is Jmangler [10]. AspectJ's shadow mungers are similar to Jmangler's *code transformer*, (and AspectJ's type mungers—used to implement inter-type declarations—are similar to Jmangler's *interface transformations*). They have similar power, but AspectJ does not attempt to follow Jmangler's automatic composition rules; instead, it leaves composition order in the hands of the programmer. The programming framework of Javassist [4] is also similar to AspectJ's implementation, with its traditional additions corresponding to type mungers and its new bytecode weaving portion corresponding to shadow mungers. Since both of these are explicitly meta-programming tools and deal with code directly, neither of these has the notion of runtime residuals that AspectJ's semantics requires.

9. Summary

The AspectJ compiler must fulfill two requirements of correctness and performance. Correctness means that it must faithfully implement the AspectJ language semantics. Performance requires not only that the compiler perform adequately but that the woven code must have roughly the same performance costs as would a hand-implemented cross-cutting concern.

This paper has presented the AspectJ advice weaving implementation as a mirror of the AspectJ language, from the representation of a piece of advice as an annotated method, to the representation of a join point as a region of bytecode plus residue, to the matching and implementation rules for the application of a piece of advice to a join point.

It has presented benchmarks that show the AspectJ compiler has performance comparable to Sun's javac when weaving aspects that only affect a small number of classes. When weaving concerns that crosscut the entire system it adds a significant 4x to the compile-time; however, this is an acceptable overhead for a large number of applications. Finally, we showed that the woven code for a modular logging policy captured by AspectJ has performance comparable to a tedious and tangled by-hand implementation of that same policy.

10. References

[1] The AspectJ Team. AspectJ programming guide. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html

[2] Jason Baker and Wilson Hsieh. Runtime Aspect Weaving Through Metaprogramming. AOSD 2002. Enschede, The Netherlands. April 2002.

[3] Bill Burke and Adrian Brock. Aspect-Oriented Programming and JBoss. http://www.onjava.com/lpt/a/3878.

[4] Shigeru Chiba and Muga Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. GPCE 2003. Springer-Verlag, 2003

[5] Joseph Gradecki and Nicholas Lesiecki. Mastering AspectJ: Aspect-Oriented Programming in Java. John Wiley and Sons. 2003.

[6] JDT core compiler version 2.1.1. http://eclipse.org

[7] Gregor Kiczales., et al. An Overview of AspectJ. *15th European Conference on Object Oriented Programming (ECOOP).* Springer. June 2001.

[8] Gregor Kiczales et al. Getting Started with AspectJ. Communications of the ACM. Volume 44 , Issue 10 (October 2001).

[9] Ivan Kiselev. Aspect-Oriented Programming with AspectJ. SAMS. 2002.

[10] Günter Kniesel, Pascal Costanza, Michael Austermann. JMangler - A Framework for Load-Time Transformation of Java Class Files. IEEE Workshop on Source Code Analysis and Manipulation (SCAM), colocated with International Conference on Software Maintenance (ICSM). November 2001.

[11] Ramnivas Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Company. 2003.

[12] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification 2nd Edition. Addison Wesley. 1999.

[13] Martin Lippert. An AspectJ-Enabled Eclipse Core Runtime. OOPSLA 2003 Poster Session. Anaheim CA. October 2003.

[14] Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the International Conference on Software Engineering (ICSE).* ACM, Limerick, Ireland. 2000.

[15] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. AOSD 2003, Boston, MA. 2003.

[16] Xalan-2.5.1. http://xml.apache.org/xalan-j.

[17] XSLTMark-2.1.0. DataPower Technology. http://www.datapower.com/xsltmark.