Optimising AspectJ *

Pavel Avgustinov[1], Aske Simon Christensen[2], Laurie Hendren[3], Sascha Kuzins[1],
Jennifer Lhoták[3], Ondřej Lhoták[3], Oege de Moor[1], Damien Sereni[1],
Ganesh Sittampalam[1], Julian Tibble[1]

[1] Programming Tools Group
Oxford University
United Kingdom

[2] BRICS
University of Aarhus
Denmark

[3] Sable Research Group
McGill University
Montreal, Canada

http://aspectbench.org

Abstract

AspectJ, an aspect-oriented extension of Java, is becoming increasingly popular. However, not much work has been directed at optimising compilers for AspectJ. Optimising AOP languages provides many new and interesting challenges for compiler writers, and this paper identifies and addresses three such challenges.

First, compiling *around* advice efficiently is particularly challenging. We provide a new code generation strategy for *around* advice, which (unlike previous implementations) both avoids the use of excessive inlining and the use of closures. We show it leads to more compact code, and can also improve run-time performance. Second, woven code sometimes includes run-time tests to determine whether advice should execute. One important case is the *cflow* pointcut which uses information about the dynamic calling context. Previous techniques for *cflow* were very costly in terms of both time and space. We present new techniques to minimise or eliminate the overhead of *cflow* using both intra- and interprocedural analyses. Third, we have addressed the general problem of how to structure an optimising compiler so that traditional analyses can be easily adapted to the AOP setting.

We have implemented all of the techniques in this paper in *abc*, our AspectBench Compiler for AspectJ, and we demonstrate significant speedups with empirical results. Some of our techniques have already been integrated into the production AspectJ compiler, *ajc* 1.2.1.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization; D.3.4 [*Programming Languages*]: Processors—compilers

General Terms Performance, Experimentation, Languages

Keywords AspectJ, optimization, aspect-oriented programming language, cflow pointcut, around advice

1. INTRODUCTION

Aspect-oriented programming is a new programming paradigm that is rapidly growing in acceptance, in large part due to the popularity of AspectJ [4], an aspect-oriented extension of Java that is compatible with existing Java programs. Aspect-oriented programming encompasses many different language features, which can be separated into two categories: *static* features, essentially a version of open classes; and *dynamic* features. While the static features incur no substantial performance penalty, that is not necessarily the case for the dynamic features. New optimisation strategies are required to improve the performance of programs written using the dynamic features of AspectJ.

1.1 Challenges

The execution model for (the dynamic features of) AspectJ is that execution of the main program is monitored for certain programmer-specified events. When these occur, control is passed to *advice*, a special kind of method that contains the extra code to be executed. Advice can be run before or after the event that triggers it, or it can be executed "around" it (instead of it).

The monitoring of the base program to find points where advice should apply can be expensive. Of course, the AspectJ language has been designed so that most of the conditions that cause advice to be triggered can be determined statically, and so do not induce overheads. A nice explanation from the viewpoint of partial evaluation can be found in [18].

However, some features of AspectJ require the insertion of dynamic checks in the base programs, resulting in performance loss. Furthermore, "around" advice, which we describe in more detail in Section 2, is a powerful language feature that is difficult to implement efficiently. It has been shown that the combination of these factors can result in substantial overhead in AspectJ programs [10]. Here, *overhead* should be understood as the additional cost for matching events to advice, not the execution of advice code itself. In addition, we must consider *space overhead*. This is the size of the extra code introduced when aspects are applied to a program.

These problems highlight the need for novel aspect-specific optimisation strategies.

1.2 The *ajc* Compiler

The reference implementation of AspectJ, originally developed by the language's designers, is the *ajc* compiler [4], maintained as part of the Eclipse project. The *ajc* compiler has been designed with compilation speed in mind, and supports incremental compilation. These design aims reduce the opportunities for optimisation, and indeed *ajc* performs only a small amount of intraprocedural analysis and no whole-program analysis. This results in substantial over-

* This work was supported, in part, by NSERC, EPSRC and IBM. Thanks to Jingwu Li for his prototype implementation of around inlining to static methods.

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006…$5.00.

head in programs produced by *ajc* when using certain features of AspectJ [10].

1.3 The AspectBench Compiler (*abc*)

The observation that aspects can introduce substantial overheads has prompted us to build a new compiler for the AspectJ language, the AspectBench Compiler (*abc*) [1, 5], which is freely available under the GNU LGPL. While *abc* uses a weaving strategy similar to *ajc* in many ways, by contrast its design was motivated by two goals: to be *extensible* (so that new features can be added to the input language) and provide a powerful *analysis and optimisation framework*. This second feature, which aims to reduce the overheads of AOP, is the main focus of this paper.

To achieve this, *abc* uses the Soot program analysis and transformation framework [24] as a backend. Soot provides a typed, three-address intermediate representation (called Jimple) and a library of standard program analyses. In addition, it is straightforward to define new analyses in Soot, both intraprocedural and interprocedural. In particular, the Jimple IR is extremely well-suited to program analysis and transformation.

1.4 Contributions

This paper is the first systematic study of the analysis and optimisation of aspect-oriented programs, in particular for the AspectJ language. We present the following novel contributions:

- A novel implementation strategy for '**around** advice'. Around advice is executed instead of the event that triggered it. Our implementation strategy avoids the use of closures in all but some pathological examples, unlike earlier implementations, which relied heavily on closures, and does not require inlining with the associated risk of code bloat. Furthermore, our implementation allows us to optionally run a postpass inliner which can selectively inline.
- A number of intraprocedural optimisations to reduce the overheads of '**cflow** pointcuts'. This feature of AspectJ is used to intercept method calls in the dynamic scope of others, and its naive implementation can be very costly.
- An interprocedural analysis to completely eliminate the overheads of using **cflow** in most cases. This analysis and the associated transformations illustrate our final contribution:
- A general technique for leveraging analyses and transformations for pure Java on AspectJ programs. The technique consists of first compiling the program naively, possibly inserting too many dynamic checks (for the applicability of advice) into the base program. We then analyse the resulting intermediate code, and reconsider the decisions to insert dynamic checks, based on the analysis results.

All these contributions have been implemented in *abc* and, at our suggestion, some have also been adopted by *ajc*. We present experiments to confirm that the above techniques can result in dramatic improvements in run time as well as code size. While this paper illustrates the techniques in the context of the AspectJ language, they equally apply to other aspect-oriented programming languages.

1.5 Outline

This paper is structured as follows. We first briefly introduce the relevant features and terminology of AspectJ in Section 2. Then, we describe the optimisations used in *abc*'s implementation of **around** advice in Section 3. Subsequently, optimisations specific to the **cflow** construct of AspectJ are given in Section 4. Section 5 summarises further efficiency improvements achieved by *abc*. Finally, we give related work in Section 6 and conclude in Section 7.

2. BACKGROUND AND DEFINITIONS

The execution events in an AspectJ program that can be monitored and associated with advice code are called ***join points***. A join point is a span of time in the execution of the program. Example join points are a call of a method, an execution of a method body, a field read or an execution of an exception handler. For any particular join point, the textual part of the program executed during the time span of that join point is called the ***shadow*** of the join point.

AspectJ contains a query language for picking out join points. Such a query is called a ***pointcut***. An ***advice declaration*** consists of an advice kind (**before**, **after**, **around**), a pointcut, and a body of code, called ***advice***. The advice is to be executed before, after, or instead of any join point which matches the pointcut. Whenever multiple pieces of advice apply at the same join point, precedence rules determine the order in which they execute.

A pointcut consists of a number of ***pointcut designators*** connected by boolean connectives. Each pointcut designator is either static (defining a set of join point shadows) or dynamic (defining a run-time condition). Some examples of static pointcuts are **call** and **execution**, which match all calls to and executions of a method matching a pattern; **within**, which matches all join points within class matching a pattern; and **handler**, which matches exception handlers. Some examples of dynamic pointcuts are **args** (a), which matches when actual arguments of a method have specified runtime types; **if** (e), which matches when the arbitrary Java expression e evaluates to true; and **cflow** (p), which matches when a join point is within the dynamic scope of a join point matched by pointcut p. In addition to testing runtime conditions, dynamic pointcut designators may also bind context values to variables to make them available to the advice code and to the code in **if** pointcuts. For instance, the **args** pointcut designator may bind the actual arguments.

AspectJ also contains standalone pointcut query constructs. The **declare error** and **declare warning** constructs take a pointcut consisting of only static pointcut designators along with a message. If the pointcut matches any join point shadow, the message is printed as a compile-time error or warning, respectively.

The process of compiling AspectJ programs is known as ***weaving***. The base program and advice declarations are woven together into one program which behaves as if the aspects were monitoring execution of the base program, and invoking the relevant advice.

Weaving is a two-step process. The first step, ***pointcut matching***, checks for each possible join point shadow in the program and each advice declaration, whether the pointcut could possibly match a join point at that shadow. If so, it constructs a ***dynamic residue*** of the runtime checks to be performed at the shadow to determine whether the pointcut actually matches.

The second step is ***advice weaving***. At each join point shadow where a piece of advice may apply, code is generated to evaluate the dynamic residue and, if it matches, to bind the context values and invoke the advice.

3. OPTIMISING AROUND ADVICE

Most advice in AspectJ programs adds some code to a join point, either before or after it. **around** advice is unique in that it is executed *instead* of each join point it applies to. It can, however, invoke the original join point at any time by using a **proceed** statement.

The following caching aspect demonstrates the usefulness of this interception mechanism. The aspect intercepts calls to a method named *foo* and only executes a call if the result is not in the cache already:

```
aspect Cache {
    pointcut methodsToCache(Object arg) :
            call(Object foo(Object)) && args(arg);
    Object around(Object arg) : methodsToCache(arg) {
        if (!cacheContains(arg))
            setCache(arg, proceed(arg));

        return getCachedValue(arg);
    }
    boolean cacheContains(Object arg) {...}
    void setCache(Object arg, Object value) {...}
    Object getCachedValue(Object arg) {...}
}
```

Like all forms of advice, **around** advice can have arguments. Each of these arguments has to be bound to an exposed context value by the pointcut expression: in the above example, **args**(*arg*) binds the argument of *foo* to *arg*. A **proceed** statement looks like a method call with the same number of arguments as the advice, and it executes the original join point with the bound context values set to the arguments of the **proceed** call. The **proceed** call can therefore *change* the values of context values (such as method arguments).

3.1 Implementation Issues

A number of difficulties arise in the implementation of **around** advice (and particularly the **proceed** statement), which we briefly outline before giving possible solutions.

The first problem is that **around** advice can apply in multiple places within the base program. For example, consider a piece of **around** advice with pointcut **execution**(*void foo*(..)) || **execution**(*void bar*(..)). This will apply to the bodies of methods *foo* and *bar*, and any occurrence of **proceed** in the advice body will pass control back to the method that was matched. Furthermore, the *context* of the advice applications (the values of locals used in the advised statements) can certainly differ between applications to *foo* and *bar*. This polymorphic behaviour of **proceed** is the main difficulty in its implementation.

Matters are further complicated by the fact that **proceed** can occur in arbitrary places within the advice body, including local and anonymous classes. An important implication is that **proceed** statements may be executed after the control flow has left the advice body. Any context values needed for the **proceed** call must therefore outlive the execution of the advice in this case.

Finally, aspects are not restricted to observing the base program, and in fact advice can apply to other advice. In particular, a piece of **around** advice can apply to the execution of its *own* body, directly or indirectly. Such *circular adviceexecution applications* are very rare, and usually pathological and a symptom of an error in the program. It is important to observe that these can occur, however, as we shall have to treat such applications as special cases. Note that the application of **around** advice to any other advice other than itself, or to a statement within its body (but not the whole body) is not considered circular. These other cases are common, but circular applications are rare.

3.2 General Implementation

Any piece of advice, regardless of its kind, is turned into a plain Java method (the *advice method*) both by *abc* and *ajc*. The interest lies in the translation of the **proceed** statement, minding the issues described above. The polymorphic behaviour of **proceed**, coupled with the need to store execution context, motivates the use of *closures* as a default and straightforward implementation strategy.

As a preparatory step to implementing **around**, any shadow that is advised by some **around** advice is lifted into a separate method (the *proceed method*) that can be invoked by **proceed**. Note that this is necessary as shadows need not be entire method bodies.

3.2.1 Closures

We shall now give a brief description of the closure strategy.

This approach works by defining a suitable interface type (or class type) for each advice method, as follows:

```
public interface Around$1 {
    public ret-type proceed$1(arg-type arg1, ...); }
```

Then, all calls to **proceed** are translated to calls on this closure interface. Note that the arguments passed to **proceed** can be different from the arguments that were passed into the advice method.

```
public class AspectClass {
    ret-type adviceMethod$1(Around$1 closure,
                    arg-type arg1, ...) { ...
        ret-type result=closure.proceed$1(arg1”, ...);
    ... }
    ...
}
```

For each advice application, a specialised closure type is defined that implements the interface and has members for the context values, and for each advice invocation, an instance of the closure class is passed to the advice method. This closure must then call the proceed method.

```
public class ShadowClass {
    public void shadowMethod() {
        Around$1 closure=new Around$1$Impl();
        ..initialise members with context values..
        AspectClass.aspectOf().adviceMethod$1(closure, ...)
    }
}
```

The major drawback of the closure approach is performance. Each time advice is triggered, a closure object has to be allocated on the heap, which can be a significant overhead.

3.2.2 Inlining

In all cases apart from the case of advice applying to itself (either the whole advice or a statement within it), it is possible to avoid closures by duplicating the advice method for each point in the program where the advice applies. This so called inlining (in *ajc* terminology) eliminates the need for polymorphism as the **proceed** statements in this specialised advice method always invoke the same join point shadow, and the join point context can be transferred using method arguments.

While inlining may be a good optimisation in certain situations, the duplication of the advice method for every advice application can lead to code bloat and thus is unsuitable as the general approach.

The **proceed** statement is implemented in *ajc* using these two strategies, a generic strategy based on closures and an inlining strategy that is used in certain cases. We shall now describe *abc*'s approach.

3.3 Around weaving in abc

We present a novel approach for weaving around advice. Our approach is generic: the same strategy can be employed everywhere, and it does not rely on inlining. The only exception is the pathological case of circular adviceexecution applications, described previously. We will return to that case in Section 3.3.4.

When compared to the dual strategy of *ajc*, *abc*'s around weaver never performs significantly worse and in many cases performs significantly better. Particularly,

- whenever *ajc* resorts to closure creation, *abc* can be expected to produce faster code
- when the advice code is big and applies at many locations, *abc* avoids *ajc*'s code bloat
- with circular adviceexecution applications, *abc* produces fewer closure objects and hence faster code.

We shall now describe our weaving strategy in detail.

3.3.1 The Generic Implementation

Instead of creating a closure class for each join point shadow where around advice could apply, *abc* places the proceed code from all join point shadows of a class in a single static proceed method in the class. If the shadow exceeds a certain size, it is first extracted into a dedicated static method which is then called from the proceed method. This is an optimisation that prevents unreasonably large proceed methods.

Each join point shadow at which around advice applies is replaced with a call to the advice body. Into the advice body, we pass a static class ID to identify the class from which we are calling it, and a shadow ID to identify the join point shadow within the class. To implement the **proceed** call, the advice body uses the static class ID (using a switch statement) to select the class from which it was called, and calls the static proceed method in that class (note that the ID '0' is reserved for calls on a closure interface in case the weaver has to fall back on closures). The static proceed method uses the shadow ID to select the shadow whose proceed code it must execute. Because we keep the code of each join point shadow in the class where it originally occurred, there is no need to generate accessor methods for private members.

```
public class AspectClass {
    ret-type adviceMethod$1(Around$1 closure,
        int shadowID, int staticClassID, args) {
    ...
    switch (staticClassID) {
    case 0: closure.proceed$1(shadowID, args); break;
    case 1: ShadowClass.proceed$1(shadowID, args); break;
    ...dispatch to other classes to which the advice applies...
    }
}

public class ShadowClass {
    public static ret-type proceed$1(int shadowID, args) {
        switch(shadowID) {
        case 0: ... do what the first shadow did...
        case 1: ... do what the second shadow did...
        ...handle further cases...
        }
    }
    ...
}
```

3.3.2 Context and Advice Formals

In addition to calling the right piece of code we must also ensure that values are passed for free variables used by this code (the context). We describe the implementation of context passing next.

Passing context Context from the join point shadow is needed in two places. First, it is used by the code of the shadow itself. Since we have moved this code out from the original join point shadow to the static proceed method, we must pass the required context into this method. Second, the AspectJ pointcut designators **args**, **this**, and **target** allow values from the context of the join point to be bound to formal parameters of the advice, and used in the advice body. Therefore, these values must be passed from the join point shadow site to the advice method that is called.

The dynamic residue of the pointcut guarding the advice may or may not match at run time. If the residue does not match, the advice is not executed, and the static proceed method is called directly from the shadow, so the context values are passed to it directly. If the residue does match, the advice body is called from the shadow, and it may in turn call the static proceed method (if the advice contains a **proceed** call). Therefore, from the shadow, we must pass to the advice body both the context needed by the advice body itself and the context needed by the proceed code, so that the advice body can pass it to the static proceed method.

One complication is that one advice method can apply to many different join point shadows with different context values. Therefore, we add a sufficient number of parameters of each type to the method implementing the advice to cover the context at all shadows at which the advice may apply. To keep the number of parameters reasonably small, we only add parameters of the types Object, int, float, double and long, since context values of any type can be converted to one of these types to be passed into the method.

A second complication is that the context value to be bound to an advice parameter may not be known until run time. Consider the following pointcut:

```
void around(Foo x) : args(x,..) || args(..,x) ...
```

In this case, x may be bound to the first argument of a method (if it has type *Foo*), or the last argument.[1] The dynamic residues at the join point shadow determine which part of the pointcut matches, so at the shadow, at run time, we know whether the x in the advice should be bound to the first or last parameter.

A third complication is that the advice may modify the context that is to be passed to the proceed code for the shadow. The **proceed** expression in the advice body accepts the same number of arguments as there are parameters to the advice body. In the **proceed** call to the original code from the shadow, these arguments replace the context values that were bound to the corresponding advice parameters when the advice was invoked. In the above example, if we call **proceed** (null) from the advice, the code of the shadow must be executed with its first or last argument replaced with null, depending on which clause of the pointcut matched at the join point before the advice was executed. Since this binding is only known at the dynamic residue and only at run time, we must communicate it from the residue to the advice method, which then communicates it to the static proceed method. To do this, at the dynamic residue, we create a bit vector specifying the bindings, and pass it through to the relevant methods.

3.3.3 Local and anonymous classes

As we have observed previously, **proceed** statements can occur in local and anonymous classes within advice methods, and thus the **proceed** invocation can occur after control flow has left the advice method (and so its context must be stored). Our implementation strategy conveniently extends to this case, as all the necessary context is available in the advice method. We simply need to add new fields to the local or anonymous class to hold the context values and initialise these fields when the classes are instantiated.

3.3.4 Special Cases

In the previous sections we have described the generic implementation of **around** advice in *abc*. As mentioned above, in the pathological case of circular adviceexecution this cannot be used, and

[1] *ajc* as of version 1.2.1 avoids this complication by issuing a compiler limitation error when encountering multiple binding pointcut primitives for the same advice formal.

we resort to closures instead. Furthermore, for efficiency *abc* also chooses to inline advice methods in some cases (when the advice is small and does not apply numerous times, to avoid code bloat). Here *inlining* is not taken in the *ajc* sense, but rather in the usual sense of substituting the body of the method (in this case, the advice method) at the point where it is called. We now describe these two special cases in greater detail.

Circular advice applications The execution of advice is a join point itself, so advice can apply to the execution of advice (the entire body of the advice method). These advice-on-advice applications can be expressed as a directed graph structure. When weaving into the execution of a method, the whole body of the method is moved into the corresponding proceed method. To simplify the weaving process, a topological sort is performed on the graph structure prior to weaving. This ensures that once an around advice method has been woven into, it itself is not applied to any join point shadows anymore.

Obviously, a topological sort fails in the presence of cycles in the graph, and the weaver will encounter situations where the advice method to be woven has already been woven into. This is the only case where we resort to the creation of closure objects. The semantics of AspectJ dictate that in a cyclical graph, the order of the execution of advice methods is determined by the dynamic residues before any advice is executed. Because those residues and the resulting execution order can be arbitrarily complex, we decided that closures offer a clean, general solution.

As observed previously, cycles in the advice-on-advice application graph are very rare and usually pathological. It therefore seems unnecessary to try to avoid closures under these circumstances. We have, however, strived to minimise the cost of closures; we create specialised closure classes with fields matching the types of context values, whereas *ajc* uses an expensive object array to store context values (requiring boxing of primitive types).

Inlining as an optimisation pass For very small advice, the most efficient strategy can be to inline the advice directly into the join point shadow. Inlining can also be beneficial if advice with `Object` return type applies to a join point with a simple type. In this case, the inlining can lead to the subsequent removal of the boxing and unboxing code.

Inlining is implemented in *abc* as an optimisation pass running after the around weaver.

The inlining process is implemented as a series of plain Java optimisations. The advice method is first inlined into the join point shadow as a normal Java method. A constant propagator and switch statement folder are then used to remove checks on the static class ID. Finally, the proceed method is inlined also, and its switch statement removed. Since multiple pieces of advice can apply at the same shadow, this whole process must be repeated until there are no calls left to inline. A special boxing remover pass removes unnecessary boxing and unboxing operations.

3.4 Empirical Results

To compare our strategy to *ajc*'s and to experiment with the different tradeoffs of inlining strategies, we experimented with three base programs and three aspects. The base programs are *ants*, an aspect-oriented simulation of an ants colony (following the specification of the ICFP 2004 programming contest) written by one of the authors (OdM) for use in an undergraduate course; *sim*, a discrete event simulator for certificate revocation simulation [2]; and *weka*, part of the weka machine learning library [25]. All benchmarks were run on a dual AMD Athlon 2000+ with 2GB RAM and the Sun J2SE 1.4.2 JVM.

Table 1 shows the both the execution time (in seconds) and woven code size (in bytecode instructions). For each benchmark we give results for: *abc*, using *abc*'s generic around weaver with inlining disabled; *abc* (inline), the same as *abc*, but with the aggressive postpass inliner enabled; and *ajc*, the result given by *ajc*'s around weaver (which is either closure-based or inlining, depending on the benchmark). For each benchmark, we have put the fastest time and the smallest code size in bold. Note that in most cases either *abc* or *abc* (inline) gives the fastest code and that *abc*'s code size is consistently smallest, sometimes by a significant margin.

Benchmark	Time (s)			Size (instr.)		
	abc	*abc* (inline)	*ajc*	*abc*	*abc* (inline)	*ajc*
sim-nullptr	21.9	23.9	**21.4**	**7893**	10687	10186
sim-nullptr-rec	23.6	**20.8**	124.0	**8216**	20519	10724
weka-nullptr	19.0	17.8	**16.0**	**103018**	116364	134290
weka-nullptr-rec	18.9	**18.0**	45.5	**103401**	188666	130483
ants-delayed	**17.5**	17.6	18.2	**3688**	3906	3785
ants-profiler	22.5	**19.2**	21.2	**7202**	13333	13401

Table 1. Execution Times and Code Size

To compare *abc*'s generic weaving strategy to *ajc*'s closure-based and inlining strategies, we applied two versions of the *nullptr* aspect [3] to two base programs, *sim* and *weka* (the first four lines in Table 1). The *nullptr* aspect is a very simple **around** aspect for enforcing coding standards that we found on the web when searching for examples of aspects. It simply checks for methods returning and issues error messages in the cases where null is returned. We used two different versions of the aspect, a recursive one (the original form) where the advice applies to itself and a non-recursive version where we explicitly use *!within(...)* to avoid matches within the body of the advice. The *ajc* compiler uses closure object creation for the first case (because of the recursion) and inlining for the second case, whereas *abc* uses its generic implementation for both cases.

Comparing the execution times for the non-recursive (*sim-nullptr* and *weka-nullptr*) versions to the times for the recursive versions (*sim-nullptr-rec*) and (*weka-nullptr-rec*), we can see that the execution time and code size for *abc* is almost the same, whereas *ajc* produces much slower code for the recursive versions (6 times slower for *sim* and almost 3 times slower for *weka*). For the the non recursive cases, *abc* is slightly slower than *ajc*.

From these experiments we can see that *abc* is fairly insensitive to whether the advice is recursive or not, but *ajc* pays a huge penalty when it must switch to an explicit closure strategy. *abc*'s behaviour is beneficial since programmers often make their advice recursive by accident and they need not pay a performance penalty.

There are other situations where the *ajc* weaver uses closures, which is demonstrated by the *ants-delayed* benchmark. This benchmark uses the *DelayOutput* aspect which captures calls to output methods and delays these calls until the end of the base program. This is accomplished using a local class of type *Runnable* in the advice method that calls **proceed** in its *run()* method. The *ajc* weaver has to instantiate closure objects in addition to the instances of the local class. Our weaving strategy avoids this, which explains why the *abc* results are slightly faster.

To demonstrate the adverse effects of a naive inlining strategy, we applied a profiling aspect to our ants base program (*ants-profiler*). The profiling aspect contains a relatively big piece of around advice that is applied to the execution of every method in the base program. Note that *ajc*'s inlining strategy can almost double the size of the resulting class files due to the duplication of the advice code for every advice application. Note that this increase in code size can also be observed with *abc*'s inlining strategy. However, with our weaving strategy, inlining is optional and not necessary for good performance. In *ajc*'s case, the only alternative to inlining is the use of closures with the dramatic effects on performance shown in the table. Furthermore, with our approach we can selectively inline and we are actively working on mixing different inlining strategies.

Currently we also have a prototype version of our inliner which inlines the specialized advice to static methods in a similar manner to *ajc*'s inliner, with similar performance and space usage. However, we believe that we can further improve on both the performance and space usage. Instead of putting the specialized static advice method in the class containing the shadow being advised, we put the specialized advice method in the class for the aspect. This has the advantage of avoiding visibility problems since the advice code stays in the aspect. Further, by collecting all of the specialized methods together, and by inlining the specialized proceed bodies into the specialized advice bodies, we have noticed that there are often clones produced. Thus, we believe that the space blowup can be reduced, in some important common cases, by recognizing the clones and only creating one copy of the specialized static method for each set of clones. A further optimization is to remove unneeded fetches of the aspect instance. Once the advice has been expressed as a specialized static method there is often no need for the aspect instance, and we are actively working on eliminating redundant and dead fetches of the aspect instance.

4. OPTIMISING CFLOW

In the previous section we introduced a new strategy for weaving around advice. When weaving all advice (before, after and around) the weaver must generate efficient code to handle dynamic pointcuts, i.e. those pointcuts that need a dynamic test to determine if they should execute at a join point shadow or not. In this section we concentrate on the most challenging dynamic pointcut, the **cflow** pointcut.

The **cflow** pointcut picks out join points that fall within the dynamic scope of certain events. Specifically, for any pointcut p, $\mathbf{cflow}(p)$ applies at a point in the execution of the program if p matches *some* state in the call stack at that program point. If p contains variables to be bound, then these are bound to the actual values found in the match nearest the top of the call stack. For example, the pointcut

$$\mathbf{call}(*\,foo())\,\&\&\,\mathbf{cflow}(\mathbf{call}(*\,bar(*))\,\&\&\,\mathbf{args}(x))$$

matches all calls to *foo* that occur within the dynamic scope of a call to *bar*, and binds *x* to the value of the argument of the last call to *bar*.

It is clear that the use of **cflow** pointcuts requires, in general, the insertion of dynamic tests in the program to test the current state against the condition $\mathbf{cflow}(p)$. The naive implementation of **cflow** associates a state with each **cflow** pointcut and updates this state incrementally (this implementation is described in [18]). The state is a stack of variable bindings that represents an abstraction of the call stack. Each time a join point that matches p is entered, a new item is pushed onto the stack, with all the variables in p bound to the appropriate values. When this join point is left, the top of the stack is popped. Finally, to check whether $\mathbf{cflow}(p)$ applies at a program point, it suffices to check whether or not the stack is empty; if it is non-empty then the pointcut applies and the appropriate variable bindings can be found on top of the stack.

The implementation of **cflow** (as described above and used in *ajc*) is clearly expensive, both because of the need to update the state (which happens every time p applies) and because of the dynamic tests inserted (which can, in the worst case, apply everywhere). Performance experiments confirm that the overhead introduced is substantial [10].

We introduce a number of optimisations for **cflow**, all implemented in *abc*. We first show a number of simple, intraprocedural optimisations that reduce the overhead substantially. Then, we show how the overhead can be entirely eliminated in many common cases by an interprocedural analysis. Finally, we give empirical measurements showing that the optimisations are very effective.

4.1 Intraprocedural Optimisations

The simple **cflow** optimisations focus on eliminating the more obvious inefficiencies in updating the state and checking for applicability. They are straightforward but quite effective.

4.1.1 Sharing cflow states

The first optimisation that *abc* performs is to *share* the state update and query code between related (or identical) **cflow** pointcuts whenever possible. Consider the following pointcuts:

$$call(*\,bar())\,\&\&\,\mathbf{cflow}(call(*\,foo(..))\,\&\&\,args(t,*,*))$$
$$call(*\,bar())\,\&\&\,\mathbf{cflow}(call(*\,foo(..))\,\&\&\,args(*,s,*))$$

A naive implementation would keep a stack for each **cflow** pointcut, and update and query them independently. We optimise this by observing that a single **cflow** pointcut can be written that covers the two existing instances. In this case, it is $\mathbf{cflow}(call(*\,foo(..)\,\&\&\,args(l_1,l_2,*))$ (where l_1 and l_2 are fresh variables). Note that this binds variables used in either one of the **cflow** pointcuts in the original program.

The implementation of **cflow** in *abc* attempts to *unify* each pair of **cflow** pointcuts that it finds. Unification of two pointcuts succeeds if the pointcuts are syntactically equivalent with the exception of free variables, and returns a pointcut that carries enough state to cover both pointcuts (as in the above example).

In general, this sharing of state can improve performance substantially. In fact, cases similar to the above arise frequently due to inlining of named pointcuts, a strategy used both in *ajc* and *abc*. An added benefit is that some method bodies can become smaller when this is performed (by avoiding duplication of bookkeeping code). We present empirical measurements of the performance improvements in Section 4.3.

4.1.2 Counters for cflow without bound variables

The next optimisation applies to pointcuts of the form $\mathbf{cflow}(p)$, where p does not bind any values. In this case, the state of the $\mathbf{cflow}(p)$ reduces to a stack of empty sets of variable bindings. In *ajc* 1.2, this is represented by a stack of arrays of length 0.

We improve on this in the obvious way, by replacing the stack with an integer counter that is incremented and decremented when p is entered and left respectively. This avoids repeated allocations of empty arrays. The case of a parameterless **cflow** appears to be quite common, so this optimisation is widely applicable.

4.1.3 Reuse of counters/stacks

The final simple **cflow** optimisation is the caching of **cflow** state objects (stacks or counters). The state of a given pointcut $\mathbf{cflow}(p)$ is thread-local, as it is an abstraction of the call stack. Multiple copies are therefore kept, one for each thread, and any operation on **cflow** state (updating or checking) involves retrieving the copy valid for the current thread (in the worst case, a hash table lookup).

In general, multiple operations on the same **cflow** state can occur within the body of the same method. In fact, updates to the state of a **cflow** are always paired (the state is updated when entering and leaving a join point), so in most cases the state is retrieved at least twice in any method in which it is needed at all.

We can therefore improve on the original implementation by retrieving the appropriate state object only when it is first used in a given method, and storing it for future uses in the same method.

4.2 Interprocedural Optimisations

The optimisations that we have described above reduce the overhead associated with **cflow**, but this can still be substantial. Since the **cflow** construct depends on dynamic properties of the program in general, it is impossible to eliminate such overhead entirely. However, many uses of **cflow** can be statically determined, at least at some program points. To take a simple concrete example, the pointcut **cflow**(**call**(∗ *foo*())) matches all points in the execution of the program within the dynamic scope of a call to *foo*. It is possible to determine statically that some program points can *never* be in the dynamic scope of *foo*, and that some program points *always* execute in its scope. At each such program point the **cflow** pointcut is statically known to be true or false, so the dynamic check can be eliminated. In addition, eliminating such dynamic matching code can allow the compiler to eliminate some of the state-updating code for this **cflow** (if its effects can no longer be observed after dynamic checks are removed).

Our empirical results in Section 4.3 show that **cflow** pointcuts can indeed be statically determined in almost all cases we have encountered to date. We will now describe the analysis used in *abc* to achieve this.

The idea for this analysis was introduced in [20] for a simple procedural language. The analysis has been adapted to the much wider context of AspectJ and implemented within *abc*. It requires an interprocedural analysis, but has two substantial advantages:

- It eliminates the overhead for **cflow** completely in many common cases, and
- in those cases, it allows **cflow** to be used in constructs that require static pointcuts (such as **declare warning**).

4.2.1 Analysis in abc

One of the design goals of *abc* was to make it possible to analyse the code being woven, and use the analysis results to optimise the weaving process to produce more efficient code. In particular, we wanted to be able to leverage the many analyses existing for Java code, without having to rewrite all of them to be specific to AspectJ. This is, in fact, not merely a matter of convenience: it is very hard to work out (for example) the control flow prior to the weaving of advice, because an analysis would have to take into account complex advice precedence rules, 11 different join point types, the option to change the arguments of a call via **proceed**, and so on. It follows that in general, an interprocedural flow analysis before weaving needs to contain all of the complexities of the pointcut matching and advice weaving in an AspectJ compiler. Indeed, others who have studied the static analysis of aspect-oriented programs [19, 26] (for the purpose of property checking), were forced to make strong restrictions in their implementation to circumvent these complexities when doing a direct analysis prior to weaving. Our aim, however, is to cover the full AspectJ language.

For these reasons, *abc* includes a hook to perform analyses on the Jimple code produced immediately *after* weaving, optimise the naive weaving instructions originally produced by the matcher, and then repeat the weaving process on the original code using the optimised weaving instructions. Because the woven code being analysed has no AspectJ-specific constructs, it is possible to apply standard analyses already in Soot. Of course, we also implement analyses and optimisations specific to AspectJ, but these are greatly simplified by being able to use the results of Java analyses.

The structure of the *abc* backend which makes these analyses and optimisations possible is shown in Figure 1. In normal operation, the phases are executed from top to bottom: matcher, weaver, bytecode generator (depicted in the middle column of the figure). We now discuss each of these components in more detail.

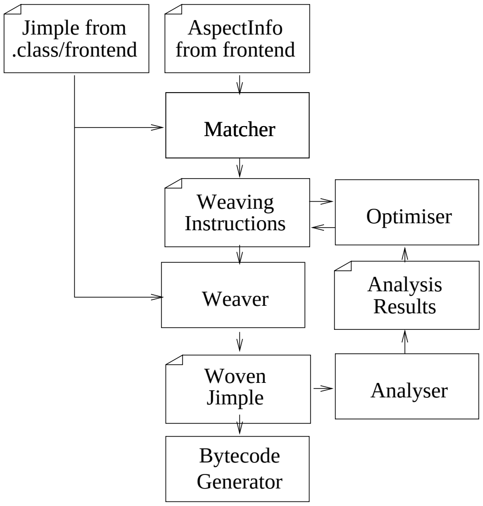

Figure 1. Reweaving in the *abc* backend

The *matcher* takes as input information about all the pointcuts and advice from the frontend, as well as the Jimple intermediate representation of the bytecode. The matcher produces a set of weaving instructions which specify where in the code they should be woven. A weaving instruction thus consists of a position in the code (which may be a single statement or a whole region of code), a piece of advice (which is represented as a regular method), and a representation of the *residue*. Often that residue is *alwaysmatch* or *nevermatch* — which means no runtime test needs to be generated. Occasionally the residue represents a truly dynamic test, for example a boolean expression for **if** pointcuts, a type instance test for **args**, or a test on the relevant stack for **cflow**.

The *weaver* executes these instructions, producing woven Jimple, where the dynamic tests and calls to the advice methods are inserted into the existing Jimple. The woven Jimple code thus corresponds to pure Java, with no aspect-oriented constructs at all. The standard *code generator* of Soot [24] can therefore translate woven Jimple to bytecode. As part of this code generation pass, we also perform simple intraprocedural optimisations (such as common subexpression elimination and constant propagation) to clean up the results of weaving.

Let us now consider how this architecture, where the matcher and weaver have been separated, facilitates analyses and optimisation: in Figure 1, this is illustrated by the boxes in the right-hand column. Because woven jimple has no aspect-oriented features, we can leverage standard Java analyses on woven Jimple, for instance the construction of a call graph: there is no need to adapt standard analyses for AspectJ. As mentioned earlier, this is very important, because it is (for example) extremely hard to work out control flow in the presence of advice prior to the weaving process. The result of such analyses is then fed into an optimiser that operates on the weaving instructions. Typically the optimiser turns dynamic residues (which would result in runtime tests when woven) into the static residues *alwaysmatch* or *nevermatch*. The weaver can then be invoked again, producing better woven code, where certain dynamic tests that always succeed or always fail no longer appear. Note that this mechanism requires saving a copy of the original Jimple code prior to the original weaving pass, which *abc* does.

So far, we have implemented an interprocedural **cflow** analysis and a thisJoinPoint escape analysis, but the approach is general;

other analyses (and corresponding optimisations) can be added to the boxes labelled Analyser and Optimiser in Figure 1.

It is possible that an analysis will produce more precise results if executed not on the naively woven Jimple, but on Jimple woven using optimised weaving instructions produced by an earlier pass of the analysis. Therefore, *abc* allows the Weaver-Analyser-Optimiser feedback loop to be repeated if desired.

4.2.2 Call Graph

Estimating which shadows may or must be in the **cflow** of other shadows requires a call graph approximating which methods may be called from which call sites. We base our analyses on a conservative call graph: every method invocation possible at run-time must be included in the call graph. Call graph construction for object-oriented languages like Java has been the subject of a significant amount of research (*e.g.* [6,9,22,23]). Rather than reinvent the wheel, we construct call graphs using Paddle, a successor of Spark [13, 14], the points-to analysis and call graph construction framework available in Soot [24].

In Java, most method calls are virtual, meaning that the method invoked depends on the run-time type of the receiver object. The treatment of virtual calls is one of the key features distinguishing different call graph construction algorithms. The Paddle framework allows us to experiment with call graphs constructed using algorithms ranging from CHA [9], which conservatively assumes that receivers could have any type admitted by their declared type, to using a subset-based points-to analysis to compute possible run-time receiver types. We used the latter option in what follows: it turns out that the extra time spent to create a precise call graph significantly reduces the time for the interprocedural **cflow** analysis we describe below.

Some applications of call graphs, such as devirtualisation, only require call edges for explicit invoke instructions present in the code. However, because methods invoked implicitly by the VM are defined to be in the **cflow** of their calling context, our call graph must include these implicit calls. In particular, we include implicit calls to static initialisers [17, section 2.17.4], calls through the `PrivilegedAction` interface, and implicit constructor calls by the `Class.newInstance` method. For the latter, the user provides a list of all classes that may be instantiated using reflection. To ensure that this list (and our call graph) is complete, we insert code into methods not reachable in the call graph to abort execution and alert us to the error. Paddle handles these tricky but important details for us; we do not need to consider them explicitly in our **cflow** analysis.

One type of implicit method invocation which we specifically exclude from the call graph used for **cflow** analysis is the invocation of the `run` method of newly created threads. In *abc*, we strive to be consistent with the AspectJ language as specified by the *ajc* implementation. In *ajc*, **cflow** stacks are maintained separately for each thread, so the code executed by a thread is not considered to be in the **cflow** of the code that created the thread. Paddle anotates each call graph edge with a kind, which we use to detect and remove edges of this unwanted kind.

4.2.3 Interprocedural cflow analysis

Desired optimisation The customary implementation of a **cflow** pointcut expression **cflow**(p) incurs overhead at two kinds of shadows. First, at each shadow matching *p*, a **cflow** stack is pushed and popped to indicate when we are in the dynamic scope of the **cflow**. We denote these shadows with the term *update shadow*. Second, at each shadow where the **cflow** pointcut could possibly match, we insert a dynamic residue to test whether the **cflow** stack is non-empty. We denote these shadows with the term *query shadow*.

We wish to perform two kinds of optimisation. First, if we can determine **cflow** stack emptiness at a query shadow statically, we can remove the dynamic residue at the query shadow, and possibly other code that becomes unreachable. Second, if we can prove that a **cflow** stack update operation will not be observed by a stack query within the dynamic scope of an update shadow, we can remove the stack update operations at the update shadow.

Analysis information required For each **cflow** stack *st* in the program, we define two kinds of sets of instructions to be computed, **mustCflow**(st), and, for each update shadow *sh* updating *st*, **mayCflow**(sh). **mayCflow**(sh) contains every instruction *i* in the program such that when *i* is executed, we may be in the dynamic scope of *sh*. That is, *i* may execute after the push operation of *sh* has been performed, but before the corresponding pop operation has been performed. **mustCflow**(st) contains every instruction *i* such that whenever *i* is executed, we must be in the dynamic scope of some update shadow *sh* updating *st*.

Whenever a query shadow (on stack *st*) is not in **mayCflow**(sh) for any update shadow *sh* (on the same stack *st*), we replace the dynamic test with a constant false pointcut expression.[2] Any query shadow in **mustCflow**(st) is replaced with a constant true pointcut expression.

In addition, we calculate a subset **necessaryShadows**(st) of update shadows whose effect may be observed at a query shadow. Each update shadow $sh \in$ **necessaryShadows**(st) satisfies two conditions. First, some query shadow *qsh* that has not been resolved statically may occur in the dynamic scope of *sh* (*i.e.* $qsh \in$ **mayCflow**(sh)). Second, *sh* may occur outside the dynamic scope of all update shadows of *st* (*i.e.* $sh \notin$ **mustCflow**(st)). This second condition enables us to mark as unnecessary those update shadows at which the stack is always already non-empty.

The optimisations become more complicated when the **cflow** binds arguments because, in this case, each query shadow not only tests whether the stack is non-empty, but also observes the entry at the top of the stack. We can still resolve statically those query shadows not in **mayCflow** (sh), since we know that the stack would always be empty when they are executed. However, at the query shadows where we know the stack is non-empty, we must keep the dynamic residues which read the entry from the stack. In addition, we can no longer remove update shadows just because they are in the **mustCflow** of the relevant stack, because we also need the correct entry to be pushed onto the stack in addition to the stack being non-empty.

Computing analysis information The exact extent of a **cflow** shadow depends on subtle details of advice precedence and the distinction between **cflow** and **cflowbelow**, and the weaver must respect these details when weaving the **cflow** stack update operations. Because we perform the analysis on the woven code, we need not consider these details; we simply consider each **cflow** shadow to start immediately after the point where the weaver wove the **cflow** push instruction, and end immediately before the corresponding **cflow** pop instruction. We need to unambiguously classify every instruction in the method as being either within or outside the **cflow** shadow. This requires that there be no jumps into or out of the shadow, which would bypass the push or pop instruction.

Due to details of the weaving process, this requirement is always satisfied, except in the case when the argument *p* of the **cflow** expression **cflow**(p) is not entirely static, and requires a dynamic residue. In this case, the weaver generates the dynamic test at the update shadow. If the pointcut *p* does not match, we do not enter

[2] The **cflow** expression may be part of a more complicated pointcut expression. Constant folding of pointcut expressions is done in a separate phase prior to weaving.

the dynamic scope of the **cflow**, so a conditional jump skips the stack update operations. Therefore, when p is not entirely static, the instructions between the push and pop may execute within or outside the dynamic scope of the **cflow**. Therefore, none of these instructions is a member of **mustCflow**(st).

Algorithm 1 is used to compute **mayCflow**(sh). It begins with the statements in the intra-procedural shadow of *sh*. Then, it adds the statements of all methods that may be called from a statement already in the set, until a fixed point is reached.

Algorithm 1 compute **mayCflow**(sh)

```
mayCflow ← $\{i \mid i$ is in intraprocedural shadow of $sh\}$
repeat
  for all methods $m \mid \exists i \in$ mayCflow . $i$ may call $m$ do
    mayCflow ← mayCflow $\cup$ set of statements in $m$
  end for
until mayCflow does not change
```

We have implemented all of the inter-procedural **cflow** analyses using Jedd [15], an extension of Java for expressing analyses using binary decision diagrams (BDDs), which it abstracts as relations. We chose to implement the analyses in Jedd for two reasons. First, they can be expressed in Jedd concisely and clearly. As an example, Figure 2 shows the Jedd implementation of Algorithm 1. Notice that the implementation closely mirrors the algorithm. Second, although the sets computed in the analyses may become quite large, they are likely to share many similarities. BDDs make is possible to represent these large sets compactly.

```
<stmt> mayCflow(Shadow sh) {
  <stmt> mayCflow = stmtsWithin(sh);
  <stmt> old;
  do {
    old = mayCflow;
    <method> targets =
        mayCflow{stmt} <> callTargets{stmt};
    mayCflow |=
        targets{method} <> stmtsIn{method};
  } while( mayCflow != old );
  return mayCflow;
}
```

Figure 2. Jedd code implementing Algorithm 1

The set **mustCflow**(st) is computed using Algorithm 2. It first accumulates the set shadowStmts of all statements within any update shadow having no dynamic residue. Every such statement is necessarily in the **cflow**. The algorithm then starts with all statements in the program, and removes statements that can be reached from the entry points of the call graph without passing through a statement that is necessarily in the **cflow**. The statements to be removed are computed by starting with the entry points, and adding statements of methods called from the set computed so far, but excluding statements in shadowStmts, until a fixed-point is reached.

Computation of **necessaryShadows**(st) is shown in Algorithm 3. We begin with all the query shadows, and remove those known statically to be false. Unless the **cflow** binds arguments, we can also remove those known statically to be true. This leaves us with the query shadows that will be tested dynamically. The necessary shadows are now those update shadows in whose **mayCflow** any dynamic query shadow appears. Unless the **cflow** binds arguments, we can also remove those update shadows which are already in the **mustCflow** of another update shadow.

4.3 Empirical Results

The **cflow** optimisations we present in this paper have been empirically validated in two different AspectJ compilers. First, we have implemented all the optimisations in our *abc* compiler. Second, we suggested them to the *ajc* team, and they have implemented counters (Section 4.1.2) and sharing (Section 4.1.1) in *ajc* 1.2.1.

Algorithm 2 compute **mustCflow**(st)

```
shadowStmts ← $\{sh \mid sh$ updates $st$ and has no dynamic residue$\}$
mustCflow ← set of all statements
targets ← set of entry points of call graph
repeat
  targetStmts ←
    $\{i \mid \exists m \in$ targets . $i$ is a statement in $m\} \setminus$ shadowStmts
  mustCflow ← mustCflow $\setminus$ targetStmts
  targets ← $\{m \mid \exists i \in$ targetStmts . $i$ may call $m\}$
until mustCflow does not change
```

Algorithm 3 compute **necessaryShadows**(st)

```
queries ← set of all query shadows on $st \cap$
        $\bigcup\{$mayCflow$(sh) \mid sh$ updates $st\}$
if cflow does not bind arguments then
  queries ← queries $\setminus$ mustCflow$(st)$
end if
necessaryShadows ← $\{sh \mid \exists i \in queries \,.\, i \in$ mayCflow$(sh)\}$
if cflow does not bind arguments then
  necessaryShadows ←
          necessaryShadows $\setminus$ mustCflow$(st)$
end if
```

4.3.1 Benchmarks

We tested the **cflow** optimisations on benchmarks from a wide range of sources. We list the benchmarks and their sizes (non-comment SLOC) in the first column of Table 2. Figure is a demo from the AspectJ documentation. Quicksort is the example from [20] with modifications suggested by Gregor Kiczales. Sablecc is a compiler written using the SableCC compiler generator, with an aspect applied to count memory allocations in each of its phases. The base programs ants, certrevsim(sim) and weka were introduced in Section 3.4. Law of Demeter [16] is a style-checking aspect that we have applied to two code bases: Certrevsim and weka. Cona [21] is a framework for specifying and checking pre- and post-conditions using aspects. We applied it to the stack example mentioned in the paper, and to the simulator.

4.3.2 Intraprocedural optimisations

		No opt.	Intra-proc	
Benchmark	SLOC	Stacks	Stacks	Counters
figure	94	5	0	1
quicksort	72	2	0	1
sablecc	31233	2	0	2
ants	939	1	1	0
LoD-sim	1586	13	0	1
LoD-weka	3912	13	0	1
Cona-stack	291	10	0	1
Cona-sim	1942	46	0	8

Table 2. Static intra-procedural optimisation counts

In Table 2, we present the static effects of our intra-procedural optimisations implemented in *abc*. The column labelled “No opt. Stacks” shows the number of different stacks before our optimisations; the “Intra-proc” column shows the number of stacks and counters after intra-procedural optimisations have been applied. In

Benchmark	*abc*					*ajc*	
	no-opt	sharing	sharing+ counters	sharing+ counters+ reuse	+inter-proc	1.2 (no-opt)	1.2.1 (sharing+ counters)
figure	1072.2	238.3	90.3	20.3	1.96	450.5	167.7
quicksort	122.3	75.1	27.9	27.4	27.3	123.5	28.9
sablecc	29.0	29.1	22.8	22.5	20.4	29.7	24.2
ants	18.7	18.8	18.7	17.9	13.1	33.0	32.9
LoD-sim	1723.9	46.6	32.8	26.2	23.7	4776.2	35.3
LoD-weka	1348.7	142.5	91.9	75.2	66.3	2349.2	113.5
Cona-stack	592.8	80.1	41.2	27.4	23.1	1107.4	56.0
Cona-sim	75.8	75.3	73.8	72.0	73.6	76.8	69.0

Table 3. Benchmark running times (seconds)

most cases, sharing reduces the number of **cflow** stacks (or counters) significantly, often down to one. In all benchmarks except ants, all **cflow** stacks are replaced with counters. A counter cannot be used for ants because the **cflow** pointcut binds a value.

We present the benchmark running times in Table 3. The middle section lists the running times of benchmarks compiled using *abc* with **cflow** optimisations disabled (no-opt), with the intra-procedural optimisations: sharing (Section 4.1.1), counters (Section 4.1.2) and reuse (Section 4.1.3). The rightmost section lists running times when the benchmarks are compiled with *ajc* versions 1.2 and 1.2.1. Between these two releases, two of the intra-procedural optimisations presented in this paper, sharing (Section 4.1.1) and counters (Section 4.1.2), were added to *ajc* in response to our suggestions.

Using the *abc* compiler, the speedups due to our intra-procedural optimisations are very significant (up to 54-fold) not only for small benchmarks (*e.g.* figure, quicksort), but also for large benchmarks which use **cflow** (*e.g.* the LoD benchmarks). We observe similar speedups with the *ajc* compiler between version 1.2 and 1.2.1, in which intra-procedural **cflow** optimisations were added.

4.3.3 Interprocedural optimisations

Static results of our inter-procedural **cflow** analysis are shown in Table 4. The “query shadows” column shows, for each **cflow** pointcut designator (corresponding to a stack or counter), the total number of query shadows and, of those, how many the analysis determined to be unreachable, how many are determined to never or always match, and how many cannot be determined statically and therefore still require a dynamic test. The “update shadows” column shows the total number of update shadows and the number that the analysis determines to be necessary, and must remain as dynamic updates even after the analysis.

With the exception of one **cflow** pointcut designator in sablecc, the analysis was able to statically determine the outcome of all **cflow** queries, and therefore entirely remove the dynamic updates and queries of the **cflow** stacks or counters. The imprecision in the sablecc case is due to query shadows in a static initialiser; to deal with this case, we are developing a simple analysis to reduce the number of spurious static initialiser edges in our call graph.

Even though the **cflow** pointcut in ants binds an argument, we can eliminate it because it is never queried. This is because the pointcut is being used as an assertion to find an error condition. By determining that the **cflow** never matches, we have statically verified the assertion. The success of the static analysis inspired us to begin experimenting with AspectJ extensions to allow “dynamic” pointcuts such as **cflow** in “static” **declare error** constructs. This provides a way for a programmer to specify properties of the program to be checked. When the analysis cannot prove the properties at compile time, a warning is issued and a run-time check inserted.

Benchmark	Query shadows					Update shs.	
	Total	Unreach.	Never	Always	Dynamic	Total	Dynamic
figure	6	0	2	4	0	6	0
quicksort	6	0	2	4	0	3	0
sablecc	985	388	299	298	0	698	0
	985	388	332	260	5	1	1
ants	84	0	84	0	0	1	0
LoD-sim	1313	798	515	0	0	41	0
LoD-weka	7031	3501	3530	0	0	41	0
Cona-stack	16	0	14	2	0	27	0
Cona-sim	2	0	2	0	0	2	0
	3	3	0	0	0	18	0
	4	3	1	0	0	31	0
	0	0	0	0	0	2	0
	7	5	2	0	0	20	0
	0	0	0	0	0	6	0
	4	0	4	0	0	5	0
	0	0	0	0	0	3	0

Table 4. Static inter-procedural optimisation counts

We were pleasantly surprised that the inter-procedural analysis was so effective in resolving **cflow** statically. To ensure that these analysis results are indeed correct, we ran all the benchmarks with a special dynamic residue woven in to check that the static analysis results always agreed with the run-time behaviour.

The performance improvements due to the removal of update shadows, query shadows, and the code of unreachable advice are shown in the “*abc* inter-proc” column of Table 3. On benchmarks making significant use of **cflow**, both small (*e.g.* figure) and large (*e.g.* LoD), these optimisations provide large speedups, even on top of the already large speedups from the intra-procedural optimisations and the use of cheap **cflow** counters. Furthermore, when the **cflow** binds an argument, the cheap counters cannot be used, so the inter-procedural optimisations enable the removal of expensive **cflow** stacks, resulting in a 1.4-fold speedup in ants.

Recall our earlier decision to use a subset-based points-to analysis to compute possible run-time receiver types in the call graph construction. If instead we use a less precise call graph, the imprecision also affects the results of our **cflow** analysis. In particular, in the sablecc benchmark, the number of update shadows optimised stays the same, but the number of query shadows optimised changes. For the stack where there are 5 unoptimised queries with the precise call graph, this number grows to 361 with CHA. Furthermore, as we remarked earlier, the extra time spent to create a precise call graph significantly reduces the time for the **cflow** analysis. Our implementations are however not yet tuned for compile-time performance, so we defer a detailed study of these issues to further work.

5. OTHER OPTIMISATIONS

In addition to **around** and **cflow** optimisations detailed in the previous two sections, *abc* also implements numerous other small optimisations which can also improve the performance of the woven code. Other optimisations fall into three categories: (1) optimising reflective access to join points; (2) reducing the amount of boxing/unboxing; and (3) cleaning up the woven code using standard Soot optimisations. We discuss each of these categories in more detail below.

5.1 Optimising Reflective Access to Joinpoints

AspectJ supports reflective access to information about join points via *thisJoinPoint*. The information available consists of static information, such as the kind of the join point and the source location, and dynamic information such as the current values of *args*, *target* and *this*. For each join point shadow, the static part is computed once, and stored in a static final field, when the class containing the shadows is first initialised. Thus, it has quite low overhead. On the other hand, the dynamic part is much more expensive since the information is specific the particular execution of a join point, and thus a join point object is created each time the join point executes.

Since it is very common for advice to only access the static part of the *thisJoinPoint* object, the AspectJ language has a special variable *thisJoinPointStaticPart* which provides only the static information and programmers are encouraged to use this variation when they only need the static information.

Of course, it would also be nice if the compiler could automatically detect those cases when only the static part is needed and thus free the programmer from explicitly having to use *thisJointPointStaticPart*. Indeed, *ajc* already implements the lazy creation of the dynamic part and *abc* implements a similar strategy. In particular, *abc* performs a simple, conservative intraprocedural analysis to determine whether the dynamic part is needed, and if not, it replaces the use of *thisJoinPoint* by *thisJoinPointStaticPart*. In cases where we do need to keep the dynamic version, it is initialised lazily. In particular it is not constructed prior to the dynamic pointcut matching, as such construction might turn out to be in vain if the pointcut fails to match.

We have plans to further improve upon our optimisations. Currently there is only one type of join point object, so one must create the whole object, even if only one part of the dynamic context is needed. We would like to be able to specialise the join point objects based on which parts are used. Secondly, we would like to examine the impact of interprocedural analyses to more accurately determine when the dynamic part is used and which parts are needed.

5.2 Reducing boxing/unboxing

At various points in an AspectJ implementation, context information must be passed, and *ajc* often does so by passing Object arrays, and using boxing and unboxing as appropriate to retrieve the relevant context information. *abc*, by contrast, exercises a lot of care to avoid unnecessary heap allocations and boxing operations.

A particular example concerns the context passing for **around** advice, and this was discussed in detail in Section 3.3.2. Another example occurs with the parameters of intertype constructors (where an aspect injects a new constructor into an existing class). *ajc* wraps all such arguments into a single Object array, but *abc* avoids boxing arguments of primitive types, at the expense of a slightly more complex code generator. As a final example, primitively typed values are sometimes boxed for passing to advice methods to allow the same advice method to be used polymorphically. After inlining, the boxing and unboxing operations can often be removed from each join point shadow.

5.3 Standard Soot Optimisations

Unlike *ajc*, which does not contain an optimising backend, in *abc* we can take full advantage of Soot to apply standard compiler optimisations after all weaving has taken place. This is beneficial because we can produce woven code which introduces spurious instructions such as *nops*, copy statements or redundant branches, and let the standard Soot analyses eliminate those extra instructions.

One example of how this simplifies our compiler is that we introduce extra *nop* instructions to cleanly represent the beginning and ending of join point shadows. This technique works well since the *nops* we introduce have no control flow entering them or leaving them and they serve as clean places on which to define exception ranges. Thus our weaver does not need to worry about patching up control flow and exception ranges as we weave. This allows the weaver to be quite compositional and extensible. After all weaving is complete the nops are removed using the standard Soot nop eliminator which automatically fixes up the control flow and exception ranges.

One surprising result we found was that weaving aspects can generate a large number of new local variables and performing Soot's register-allocation-type packing of these variables at the Jimple level resulted in significant speedups on some popular JITs.

Finally, we also apply standard Soot optimisations during specialised passes like the around inliner. These are used to remove useless parameters, remove unused methods, simplify switches and eliminate useless casts.

6. RELATED WORK

This work is the first general study of analysis and optimisation strategies for aspect-oriented languages in general, and for AspectJ in particular. As a consequence, the amount of related work is rather sparse. There are however a number of other industrial strength implementations of aspect-orientation, and we discuss these here.

The ajc Compiler The reference implementation of AspectJ (and in fact the only other implementation of the language) is the *ajc* compiler. The weaving strategies of *ajc* and *abc* are similar, except for the optimisations described in this paper. Following the early success of our optimisations in *abc*, two of them (**cflow** counters and sharing of **cflow** stacks) have been incorporated into *ajc* 1.2.1. Further details on the implementation of weaving in *ajc* (similar to weaving in *abc* except for the optimisations described here) can be found in [11].

Other AOP Systems There are a number of other systems besides AspectJ that support the use of aspects. Perhaps the most successful of these is AspectWerkz [8]; its features are in fact very similar to those of AspectJ.[3] Aspects are however deployed using annotations or scripts, rather than in an extension of the Java language. Unlike AspectJ, AspectWerkz supports dynamic weaving: enabling new aspects at runtime, and also disabling them:

The AspectWerkz system may however also be used in off-line mode, in the same way as *ajc* or *abc*. Because of its focus on runtime weaving, AspectWerkz employs an event-based implementation of join points, where advice can register as a listener. In preliminary experiments, we have found this strategy leads to a slowdown of a factor of 9 or more compared to *ajc* or *abc*. Because of this huge gap and the different aims of dynamic weaving, we focused on the most popular static weaving system, which is *ajc*. The

[3] Indeed, since January 2005, the *ajc* and AspectWerkz projects have merged — the discussion here refers to AspectWerkz 1.0.

other leading AOP system is JBoss [12], and this employs a similar implementation strategy to AspectWerkz. It is evident that both these systems could benefit by the optimisations presented here, when used in off-line mode. For weaving at runtime, it would appear that our intraprocedural optimisations may be helpful. The same applies to efforts to support aspects in a modified JVM, as in [7].

7. CONCLUSIONS

The field of optimising compilers for AOP languages is just starting, but we believe that this area will provide many interesting problems and challenges that can be met with both existing and new compiler optimisation technology.

In this paper we have presented three main contributions to the field in the context of a new optimising compiler for AspectJ, *abc*. We have designed and implemented a new strategy for weaving **around** advice which aims to avoid both the code size explosion of a pure inlining approach and the time and space overhead of an explicit closure-based approach. Our experimental results demonstrate that this technique works very well, it is much more efficient than the closure-based approach, and produces much less code than the inlining-based approach.

Our second major contribution was to show how to reduce or eliminate the large overheads associated with **cflow**. We gave some intra-procedural techniques that are both relatively simple and very effective at reducing large overheads for the common case. These optimisations have already been adopted by the implementors of the *ajc* compiler. We then showed that we can go even further by applying inter-procedural analyses that can statically approximate dynamic **cflow** properties. Our experimental results show that in many cases we can completely eliminate the **cflow** overhead.

Finally, the implementation strategies presented here showcase a novel methodology for defining new program analyses and efficiency-improving program transformations for aspect-oriented languages. In particular, the interprocedural **cflow** analysis shows that *reweaving* is a useful technique. In reweaving, aspects are woven first naively into the base program, the resulting program is analysed and the results of the analysis are used to guide subsequent weaving phases (so that better code can be produced). In general, reweaving can be iterated multiple times. The *abc* compiler was designed specifically to support reweaving and thus can serve as a workbench for developing new optimising transformations of AspectJ.

References

[1] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.

[2] André Årnes. PKI certificate revocation. Available at `http://www.pvv.ntnu.no/~andrearn/certrev/`.

[3] R. Dale Asberry. Aspect Oriented Programming (AOP): Using AspectJ to implement and enforce coding standards. `http://www.daleasberry.com/newsletters/200210/20021002.shtml`, 2002.

[4] AspectJ Eclipse Home. The AspectJ home page. http://eclipse.org/aspectj/, 2003.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *AOSD 2005*, pages 87–98, March 2005.

[6] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA 1996*, pages 324–341, 1996.

[7] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD 2004*, pages 83–92, 2004.

[8] Jonas Bonér. AspectWerkz — dynamic AOP for Java. Available from URL: `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`, 2004.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP 1995*, volume 952 of *LNCS*, pages 77–101, 1995.

[10] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *OOPSLA 2003*, pages 149–168. ACM Press, 2003.

[11] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *AOSD 2004*, pages 26–35. ACM Press, 2004.

[12] JBoss. JBoss Aspect Oriented Programming. Home page with downloads, documentation, wiki. `http://www.jboss.org/index.html?module=html&op=userdisplay&id=developer%s/projects/jboss/aop`.

[13] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[14] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *CC 2003*, volume 2622 of *LNCS*, pages 153–169. Springer, April 2003.

[15] Ondřej Lhoták and Laurie Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI 2004*, pages 158–169, 2004.

[16] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the Law of Demeter with AspectJ. In *AOSD 2003*, pages 40–49. ACM Press, 2003.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[18] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC 2003*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.

[19] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the Twelfth International Symposium on the Foundations of Software Engineering*, pages 147–158, 2004.

[20] Damien Sereni and Oege de Moor. Static analysis of aspects. In *AOSD 2003*, pages 30–39, 2003.

[21] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA 2004 Companion*, pages 196–197, 2004.

[22] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *OOPSLA 2000*, pages 264–280, 2000.

[23] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA 2000*, pages 281–293, 2000.

[24] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC 2000*, pages 18–34, 2000.

[25] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java implementations*. Morgan Kaufmann Publishers, 2000.

[26] Jianjun Zhao and Martin Rinard. System dependence graph construction for aspect-oriented programs. Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT, 2003.